第6章　树和二叉树

王　　勇
计算机/软件学院　　大数据分析与信息安全团队
21#518　　电　话 13604889411
Email：　wangyongcs@hrbeu.edu.cn

哈爾濱工程大學
Harbin Engineering University

问题提出

- 网页链接
- 人类社会族谱
- 社会组织结构
- 源程序的语法结构树

知识点

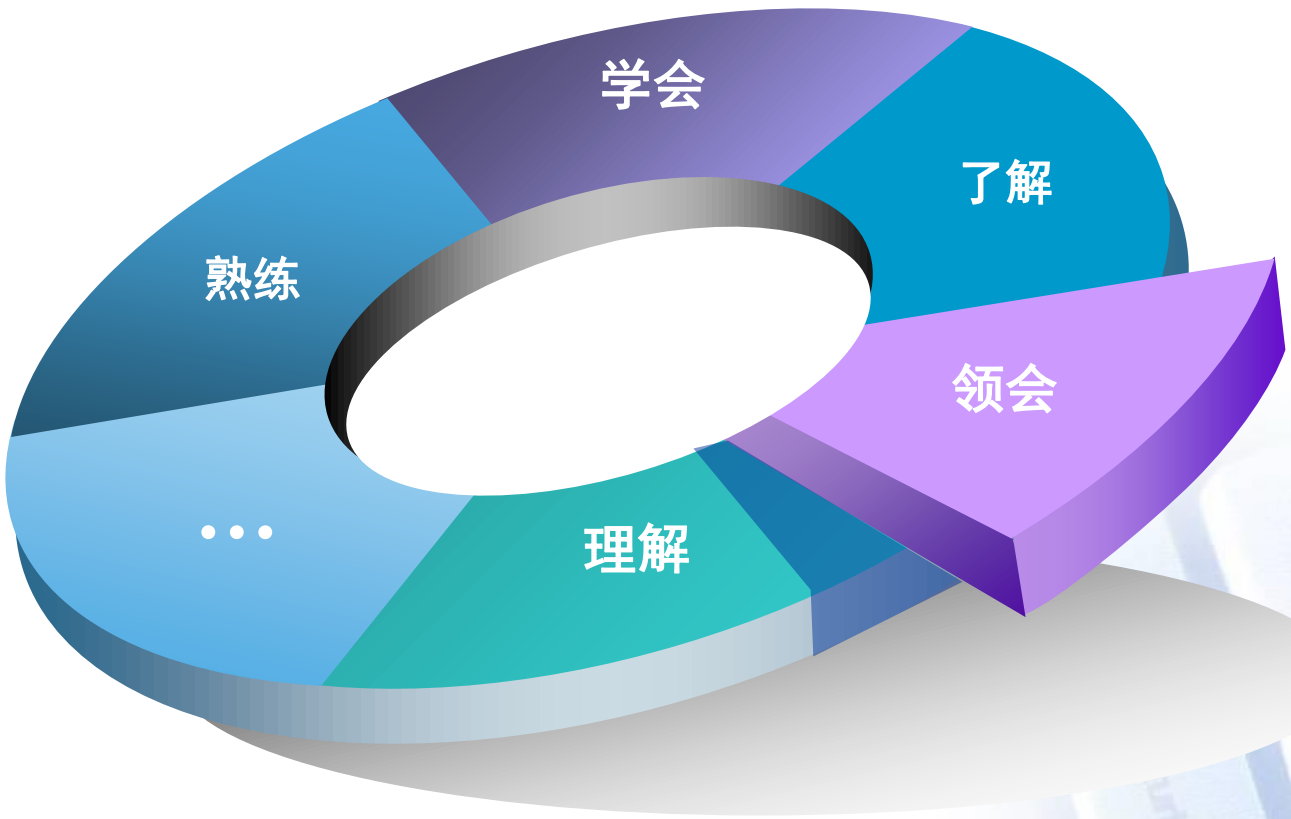
树的类型定义
二叉树的类型
二叉树的存储
二叉树的遍历
二叉树相关操作
线索二叉树
树和森林存储
树和森林的遍历
树和森林相关操作
最优树和赫夫曼编码

重 点

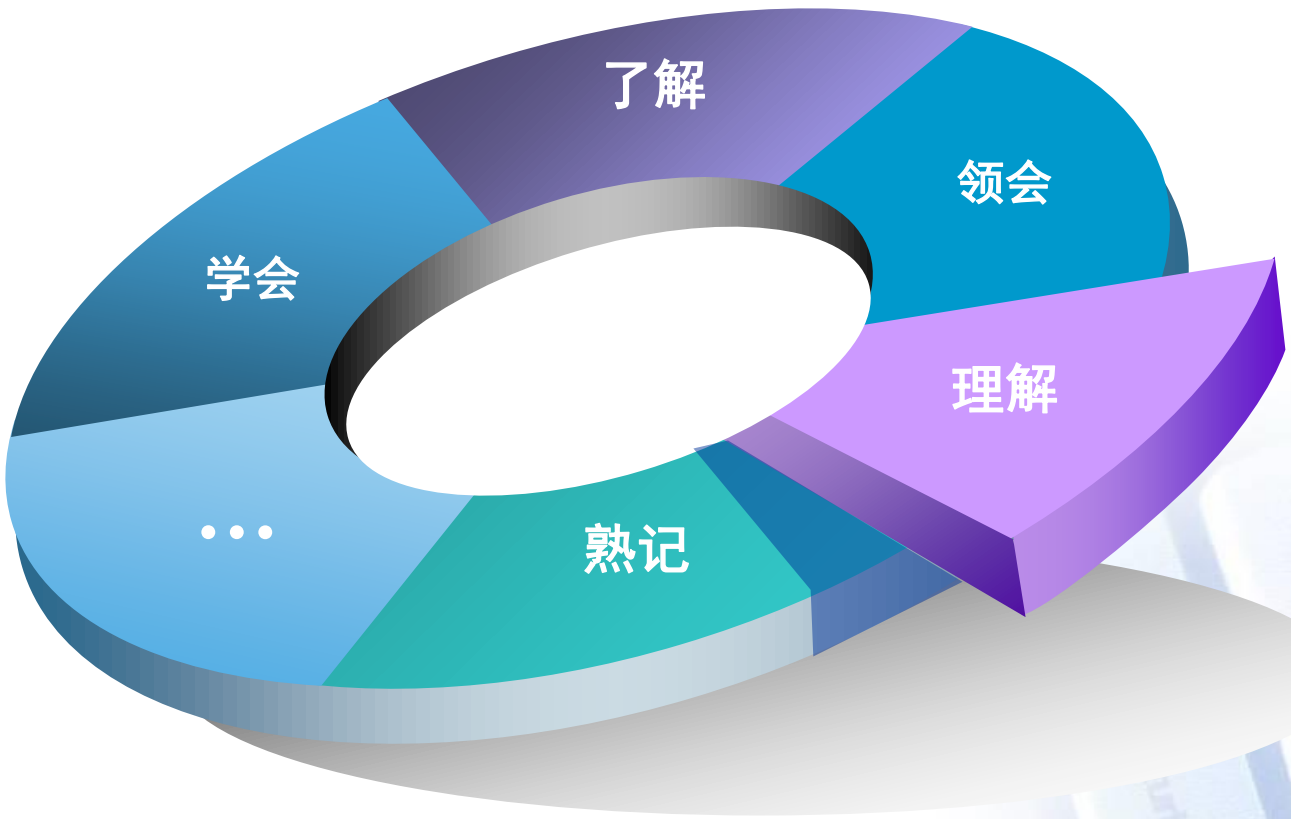
二叉树的
遍历及其应用

难 点

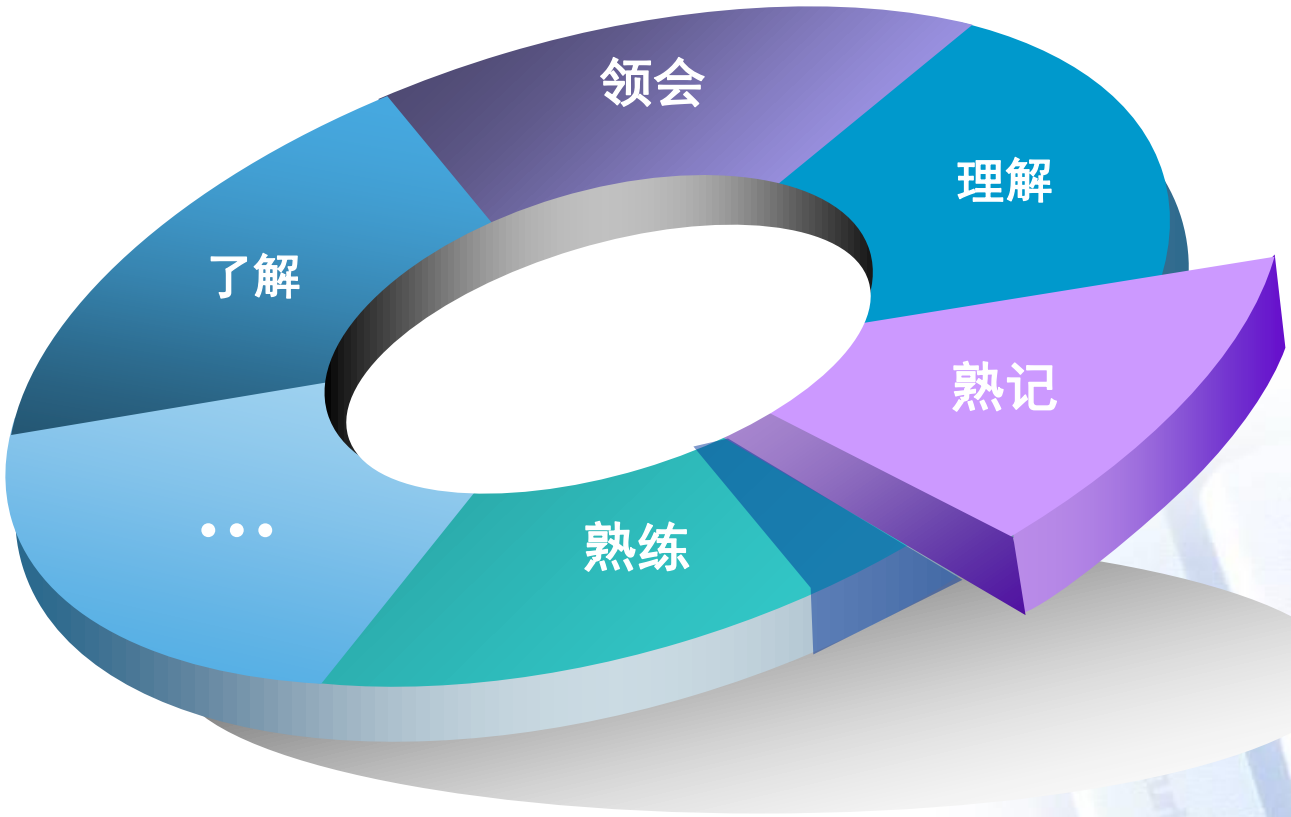
编写实现
二叉树和树
各种操作
的递归算法



树和二叉树的类型定义

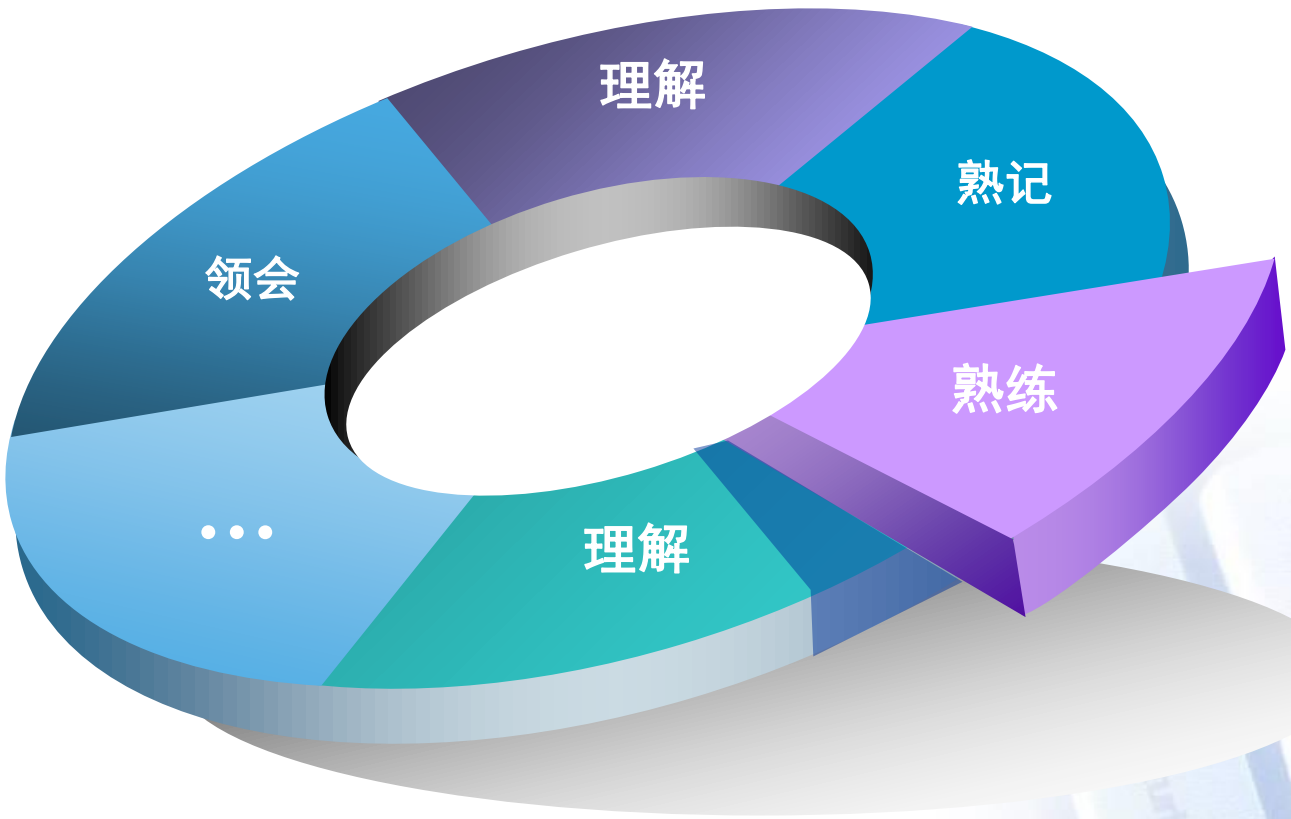


树和二叉树的结构差别

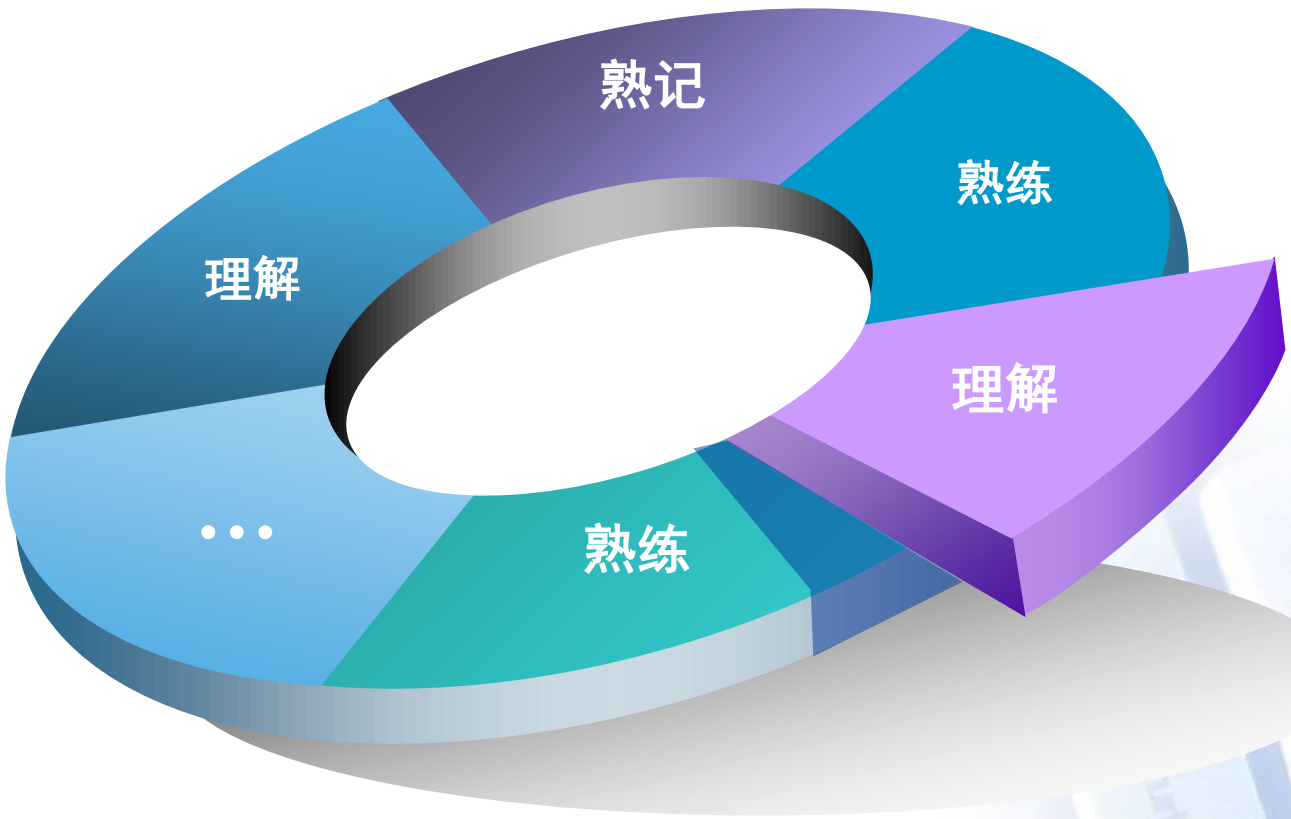


二叉树的主要性质，并掌握它们的证明方法

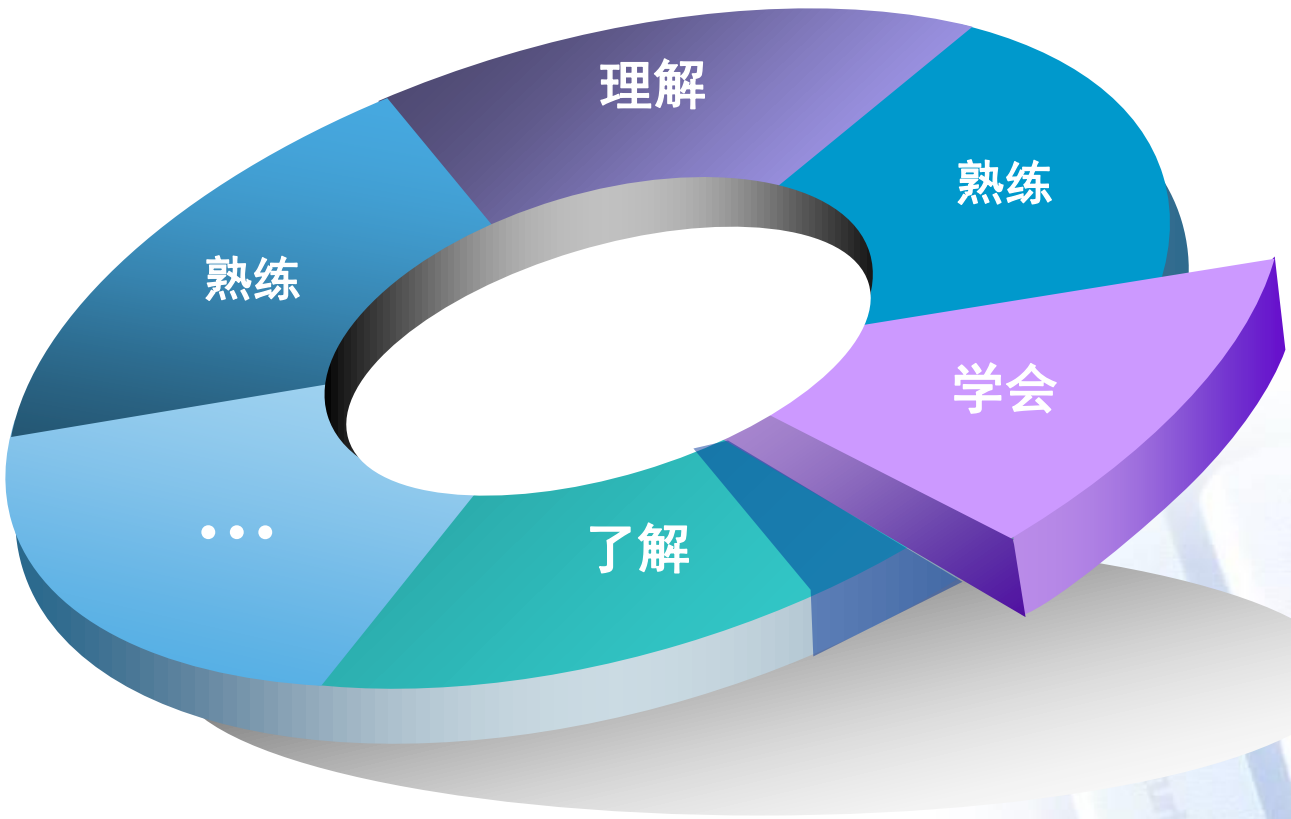
本章说明



掌握二叉树各种遍历算法，并能灵活运用遍历算法实现二叉树的其它操作

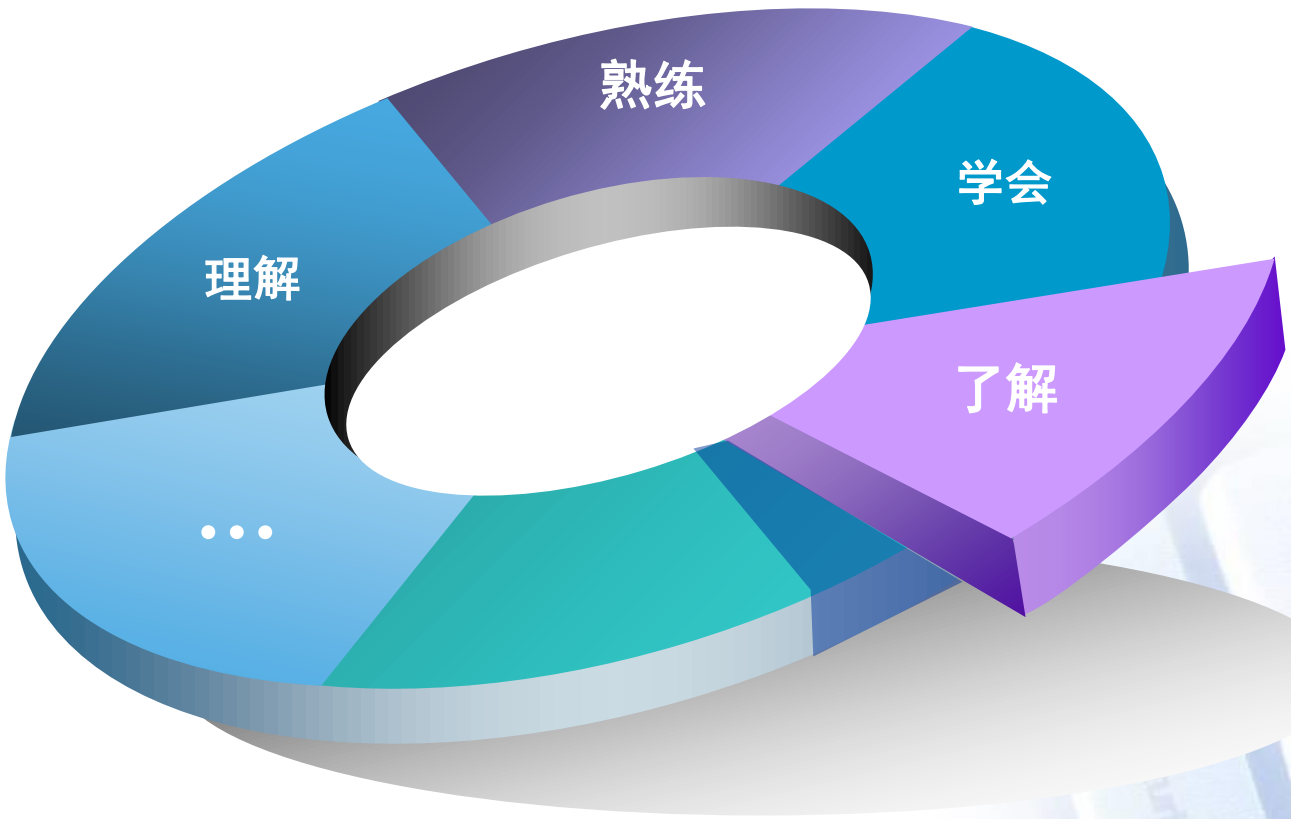


二叉树的线索化过程以及在中序线索化树上查找给定结点的前驱和后继的方法



编写实现树以及二叉树的各种操作的算法

本章说明



最优树的特性，掌握建立最优树和赫夫曼编码的方法

本章内容

定　义

树：是n(n>=0)个结点的有限集T

在任意一棵非空树中：

- 有且仅有一个特定的结点，称为树的根(root)
- 当n>1时，其余结点可分为m(m>0)个互不相交的有限集$T_1, T_2, \ldots\ldots T_m$，其中每一个集合本身又是一棵树，称为根的子树(subtree)

特　点

- 非空树中至少有一个结点——根
- 树中各子树是互不相交的集合

树是一类重要的非线性结构，是以分支关系定义的层次结构

树和二叉树

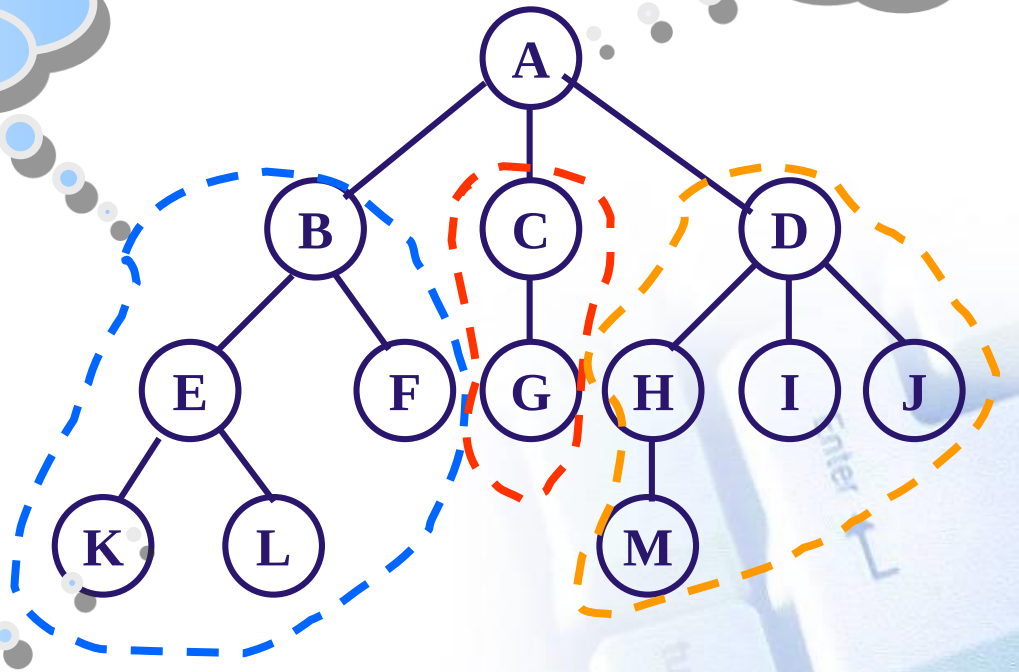
义和基本术语

只有根结点的树

A

根

有子树的树



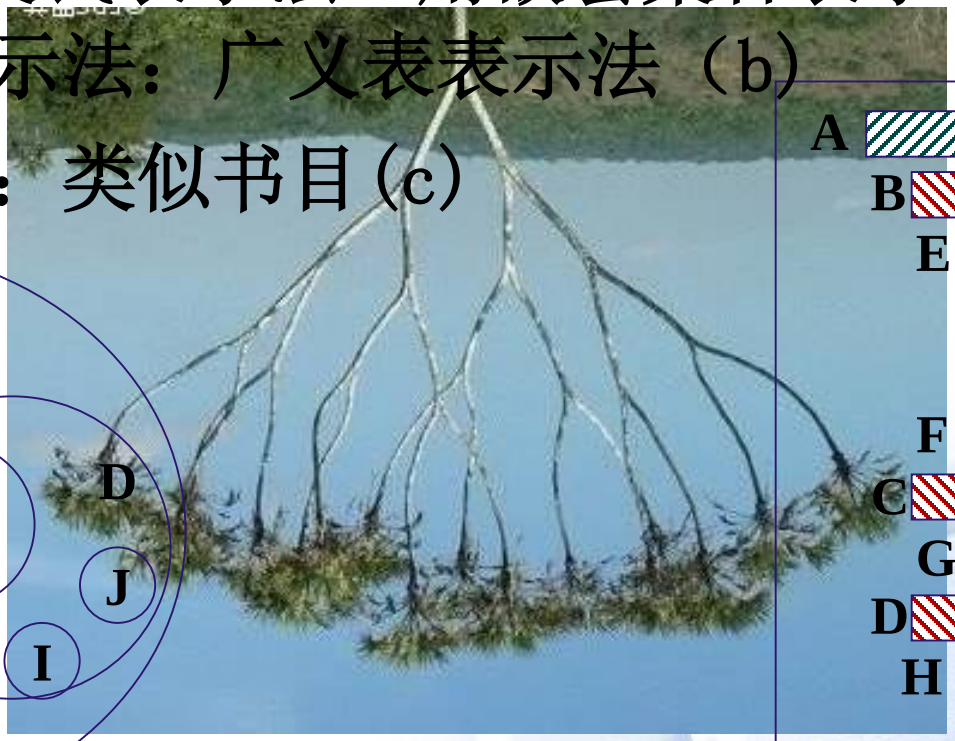
子树

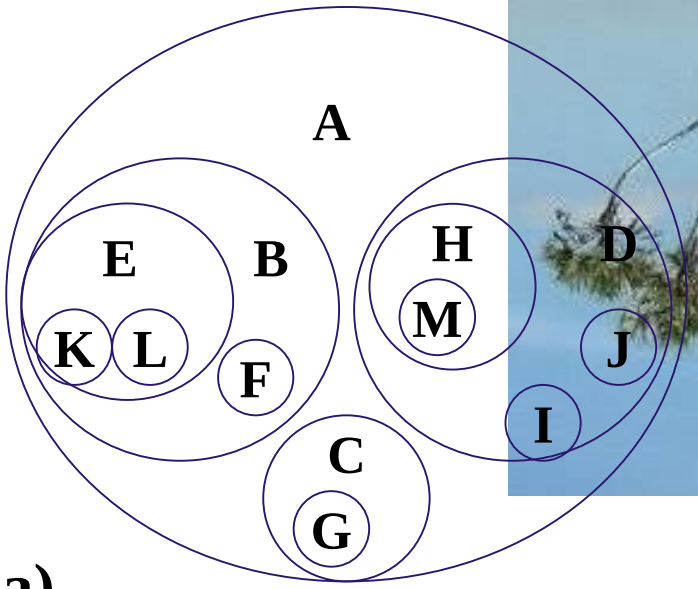
- 树的抽象数据类型的定义P118

```
ADT Tree {
    数据对象 D：具有相同特性的数据元素的集合
    数据关系 R：
    基本操作 P：
                }
InitTree(&T);
DestroyTree(&T);
CreateTree(&T,definition);
ClearTree(&T);
……
```

表示方法

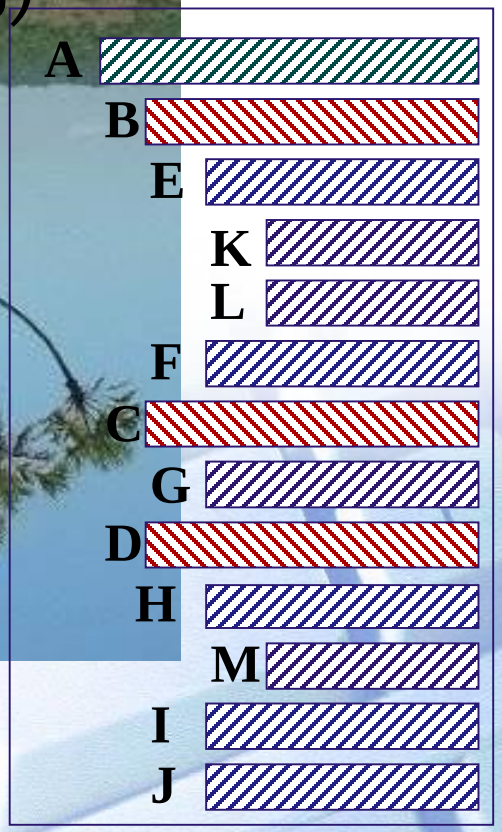
- 树形表示法：自然界倒长的树
- 文氏表示法：用嵌套集合表示(a)
- 嵌套括号表示法：广义表表示法（b）
- 凹入表示法：类似书目(c)





(a)

(A(B(E(K,L),F),C(G),D(H(M),I,J)))　(b)



(c)

术 语

- 结点(node)——表示树中的元素，包括数据项及若干指向其子树的分支
- 结点的度(degree)——结点拥有的子树数
- 叶子(leaf)——度为0的结点
- 分支结点——度不为0的结点
- 孩子(child)——结点子树的根
- 双亲(parents)——孩子结点的上层结点
- 兄弟(sibling)——同一双亲的孩子
- 树的度——一棵树中最大的结点度数
- 结点的层次(level)——从根结点算起，根为第一层，它的孩子为第二层

- 堂兄——其父母为兄弟的结点互称堂兄
- 祖先——结点的祖先是从根到该结点所经分支上的所有结点
- 子孙——以某结点为根的子树中的任一结点都称为该结点的子孙
- 有序树——树中结点的各子树从左到右有顺序，即不可以互换
- 无序树——树中结点的各子树从左至右是无次序的，即可以互换
- 深度(depth)——树中结点的最大层次数
- 森林(forest)——m(m≥0)棵互不相交的树的集合

结点A的度：3
结点B的度：2
结点M的度：0

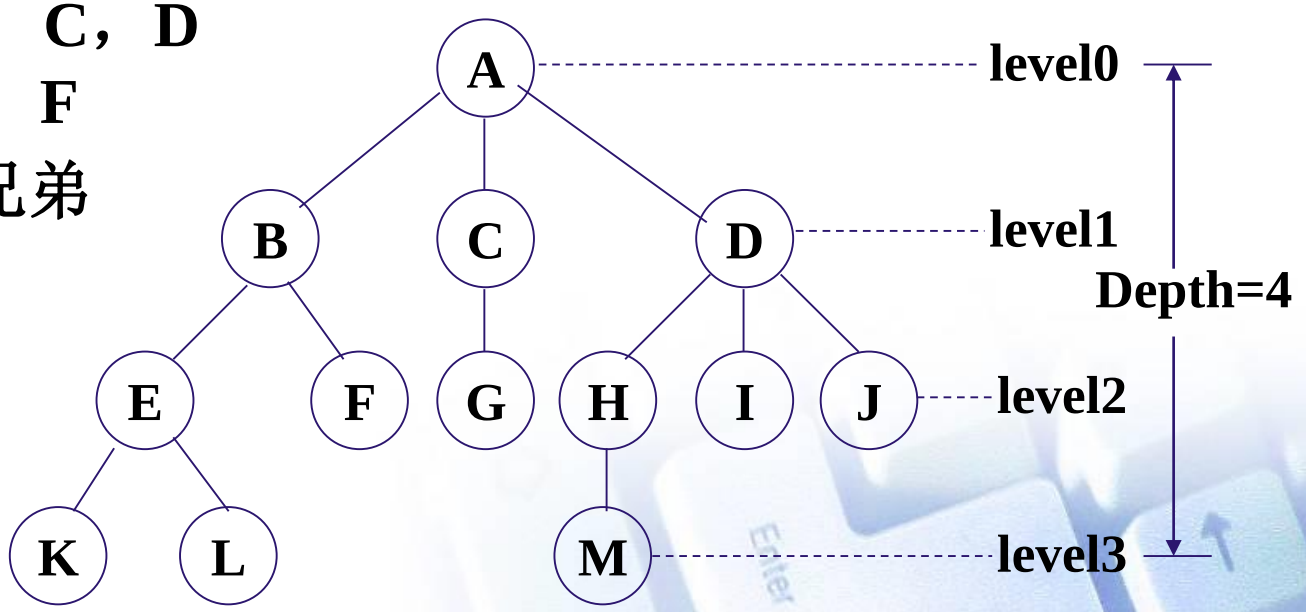
叶子：K，L，F，G，M，I，J
结点I的双亲：D
结点L的双亲：E

结点A的孩子：B，C，D
结点B的孩子：E，F

结点B，C，D为兄弟
结点K，L为兄弟

树的度：3

结点A的层次：1
结点M的层次：4



结点F，G为堂兄弟
结点A是结点F，G的祖先

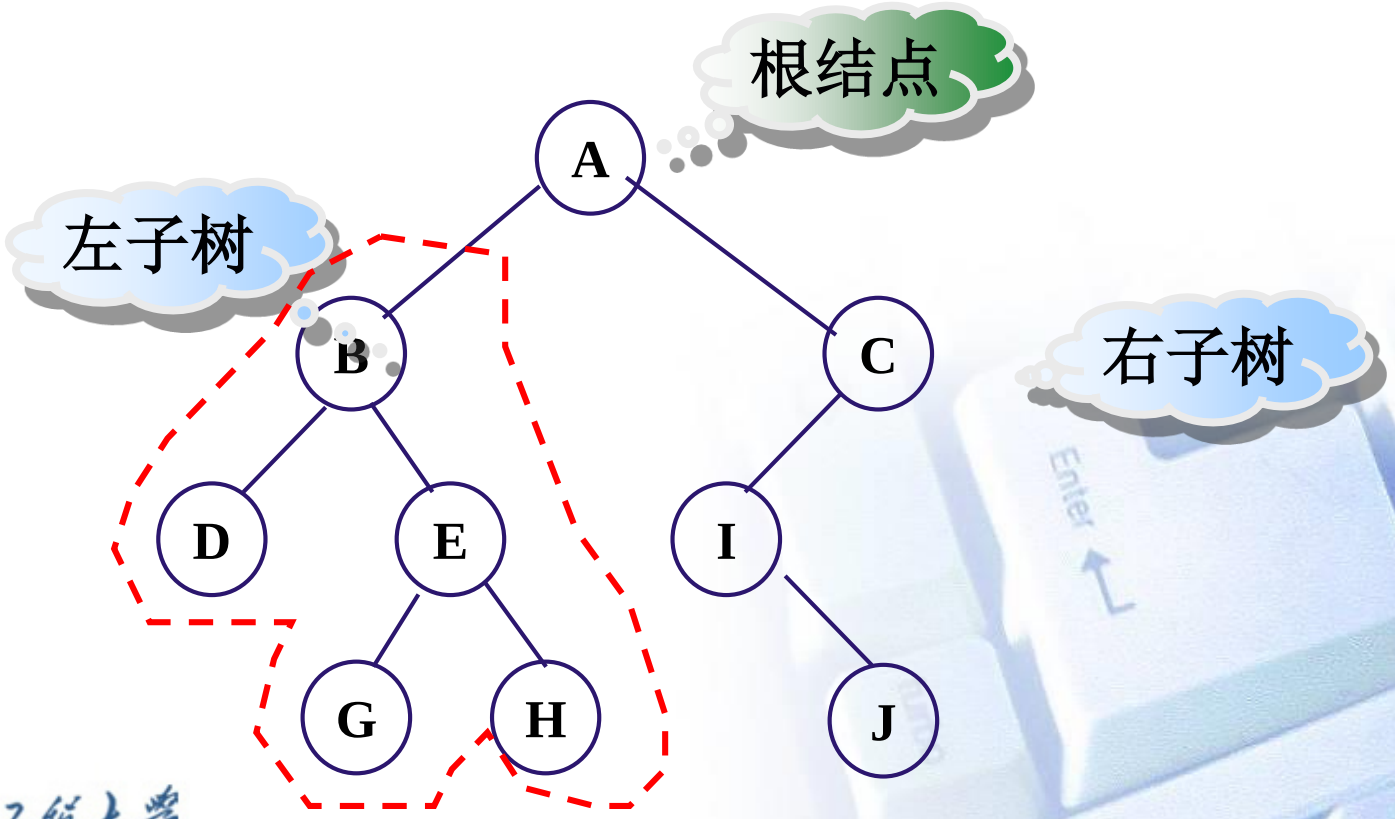
树的深度：4

树型和线性结构对照

线性结构	树型结构
存在唯一的没有前驱的"首元素"	存在唯一的没有前驱的"根结点"
存在唯一的没有后继的"尾元素"	存在多个没有后继的"叶子"
其余元素均存在唯一的"前驱元素"和唯一的"后继元素"	其余结点均存在唯一的"前驱(双亲)结点"和多个"后继(孩子)结点"

本章内容

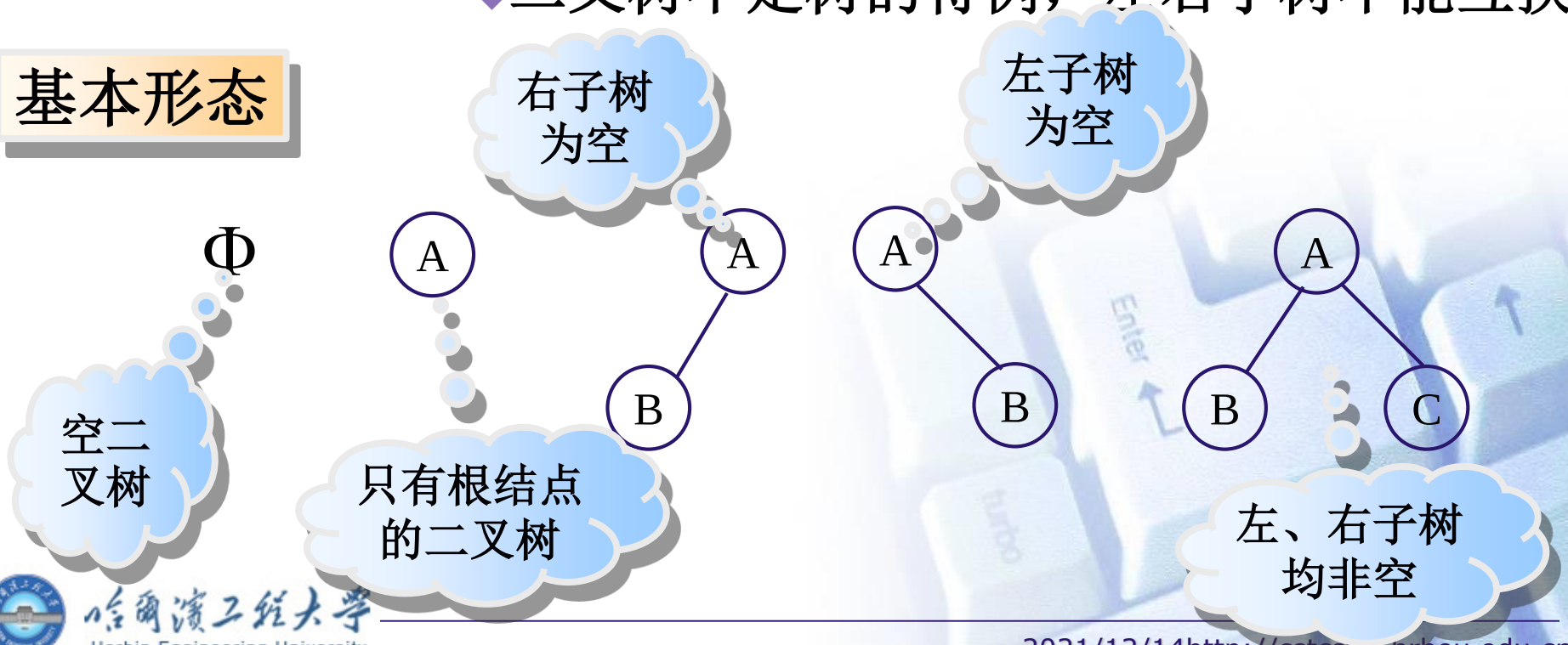
定 义 二叉树是n(n≥0)个结点的有限集，它或为空树(n=0)，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树子树构成



特　点

- 每个结点至多有二棵子树(即不存在度大于2的结点)
- 二叉树的子树有左、右之分，且其次序不能任意颠倒
- 二叉树不是树的特例，左右子树不能互换

基本形态



二叉树与树的联系与区别

- 树和二叉树逻辑上都是树形结构

- 树和二叉树的区别

  一是二叉树的度至多为2，树无此限制；二是二叉树有左右子树之分，即使在只有一个分枝的情况下， 也必须指出是左子树还是右子树，树无此限制；三是二叉树允许为空，树一般不允许为空（个别书上允许为空）。

二叉树与树的联系与区别

二叉树不是树的特例 ，并非是树的特殊情形，它们是两种不同的数据结构 。

- 二叉树与无序树不同

  二叉树中，每个结点最多只能有两棵子树，并且有左右之分。

- 二叉树与度数为2的有序树不同

  在有序树中，虽然一个结点的孩子之间是有左右次序的，但是若该结点只有一个孩子，就无须区分其左右次序。而在二叉树中，即使是一个孩子也有左右之分。

性 质1 在二叉树的第i层上至多有2^{i-1}个结点($i \geq 1$)

证明：用归纳法证明之

①i=1时，只有一个根结点，显然，$2^{i-1}=2^0=1$

是对的

②假设对所有j，($1 \leq j < i$)命题成立，即第j层上至多有2^{j-1}个结点，可证明$j=i$命题成立

③那么，第$i-1$层至多有2^{i-2}个结点

又二叉树每个结点的度至多为2

∴第i层上最大结点数是第$i-1$层的2倍，即

$$2 \times 2^{i-2}=2^{i-1}$$

故命题得证

性 质2 深度为k的二叉树至多有2^k-1个结点(k≥1)

证明：由性质1，可得深度为k 的二叉树最大结点数是

$$\sum_{i=1}^{k}(\text{第}i\text{层的最大结点数})=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

性 质3 对任何一棵二叉树T，如果其终端结点数为n_0度为2的结点数为n_2，则$n_0=n_2+1$

证明：设n_1为二叉树T中度为1的结点数

因为：二叉树中所有结点的度均小于或等于2

所以：其结点总数$n=n_0+n_1+n_2$

又二叉树中，除根结点外，其余结点都只有一个分支进入，设B为分支总数，则$n=B+1$

又：分支由度为1和度为2的结点射出，

$\therefore B=n_1+2n_2$

于是，$n=B+1=n_1+2n_2+1=n_0+n_1+n_2$

$\therefore n_0=n_2+1$

特殊形式

- **满二叉树** 一棵深度为k且有2^k-1个结点的二叉树

  特点 每一层上的结点数都是最大结点数2^{i-1}

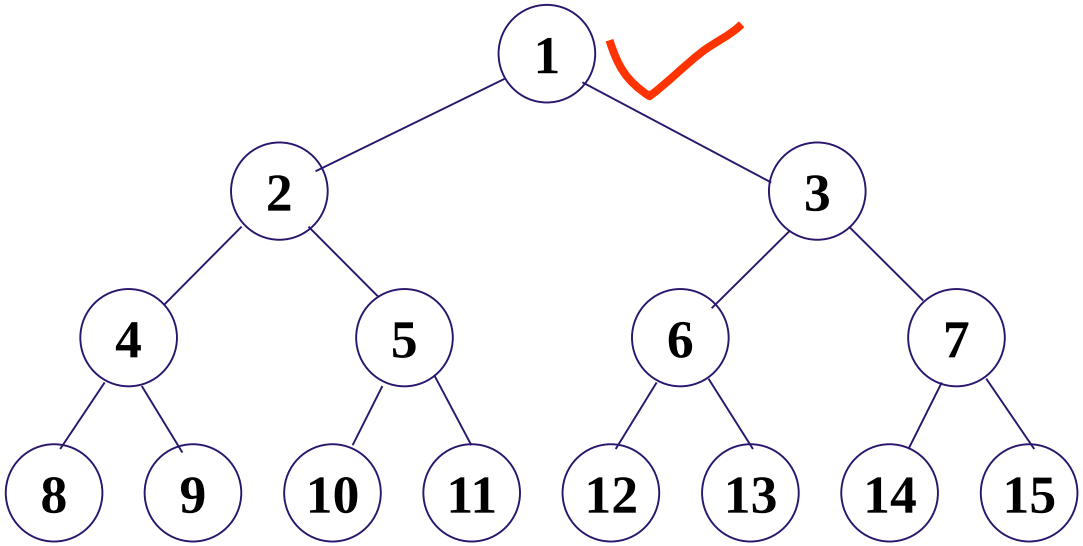
- **完全二叉树** 深度为k，有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应

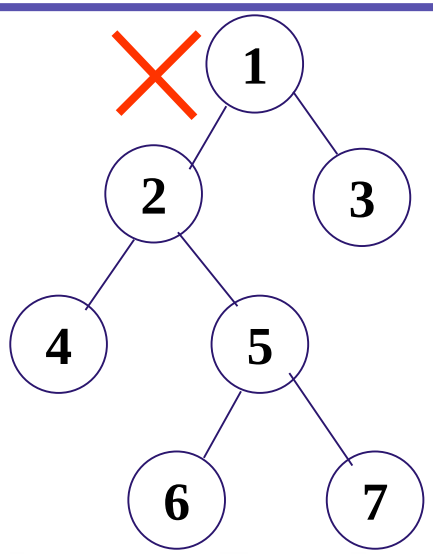
特点

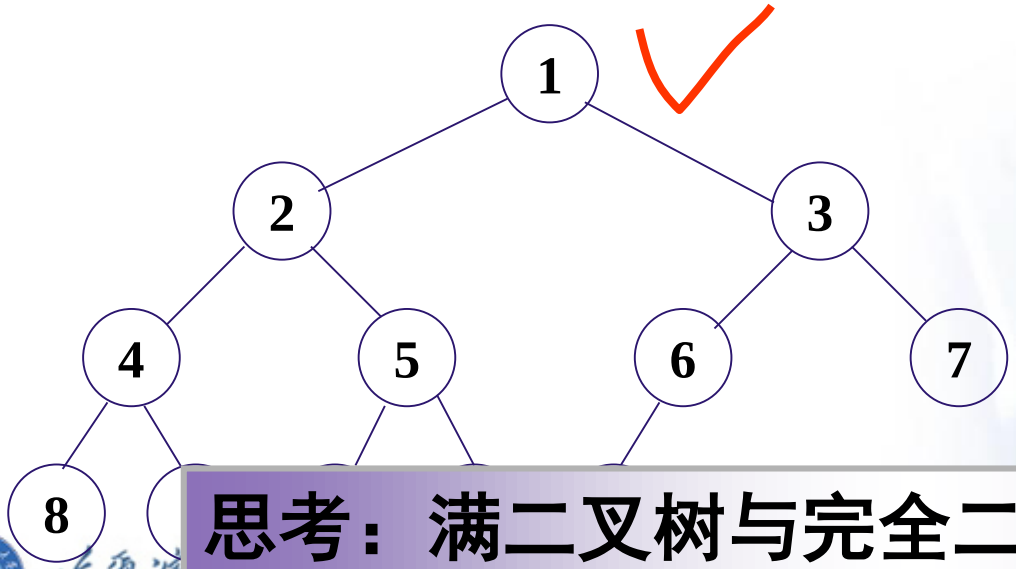
- ✓ 叶子结点只可能在层次最大(最底)两层上出现，而且基本上会出现在倒数第二层的右部
- ✓ 对任一结点，若其右分支下子孙的最大层次为l，则其左分支下子孙的最大层次必为l或$l+1$

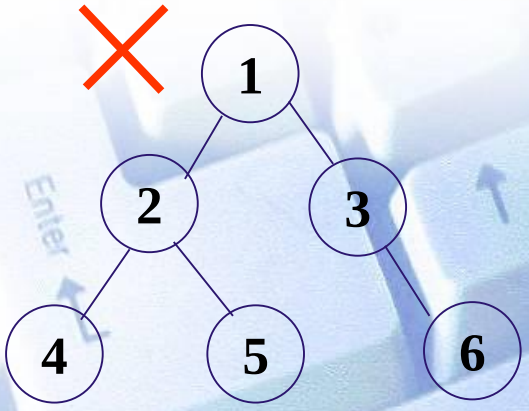
树和二叉树

二叉树









思考：满二叉树与完全二叉树的关系？

性 质4 具有n个结点的完全二叉树的深度为$\lfloor log_2 n \rfloor+1$

证明：设深度为k，则由性质2和完全二叉树定义

(k-1层)$2^{k-1}-1<n\leqslant 2^k-1$($k$层） 或 $2^{k-1}\leqslant n<2^k$

于是，$k-1 \leqslant log_2 n < k$

又因为k是整数 $\therefore k=\lfloor log_2 n \rfloor+1$

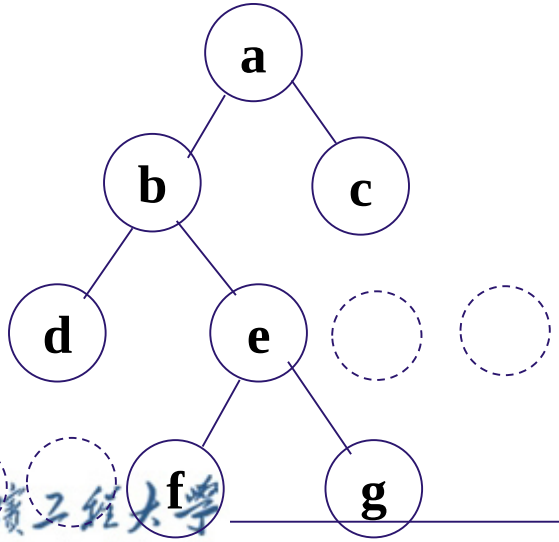
性 质5 对于一棵完全二叉树，从上到下从左至右对结点进行编号,根结点为1,则对任一结点i(1<=i<=n)，有：

- 若i=1,则结点是二叉树的根，无双亲，否则其双亲是⌊i/2⌋
- 如果2i>n，则结点i无左子女，否则，其左子女为2i
- 如果2i+1>n，则结点i无右子女，否则，其右子女为2i+1

存储结构

顺序存储

- 实现：按满二叉树的结点层次编号，依次存放二叉树中的数据元素
- 特点：
  - 结点间关系蕴含其存储位置中（i\2i\2i+1）
  - 浪费空间，适于存满二叉树和完全二叉树



1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g

浪费空间

表 示

```
#define MAX_TREE_SIZE 100
Typedef TElemType SqBiTree[MAX_TREE_SIZE];
                                    // 0号单元存储根结点
SqBiTree bt;
```

缺 点 按完全二叉树形式存储，浪费空间。

例 如 在最坏情况下，n个结点的单枝树，要占用2^n-1个元素的存储空间。

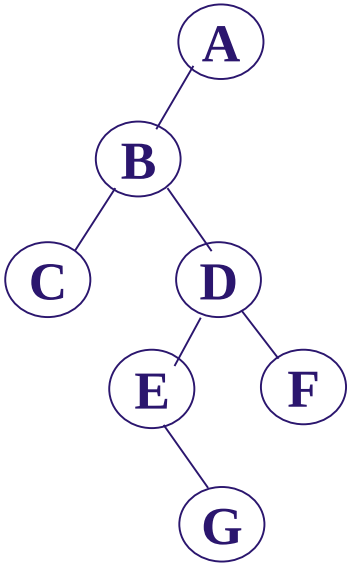
顺序存储结构适用于满二叉树和完全二叉树的存储

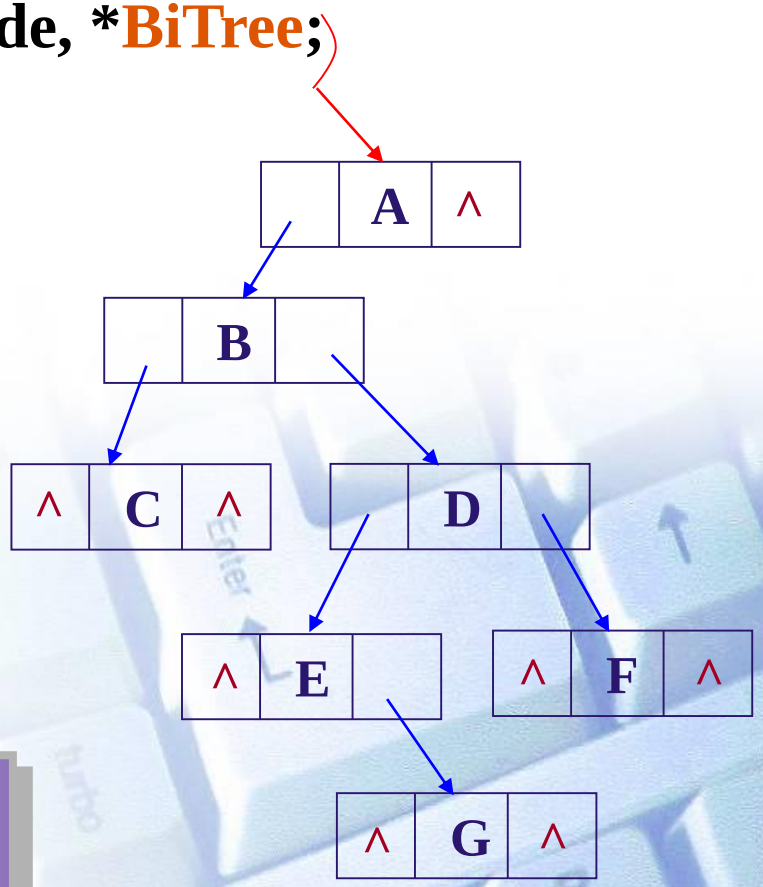
链式存储

二叉链表

lchild	data	rchild

```
typedef   struct  BiTNode
{   TElemType   data;
      struct BiTNode *lchild,  *rchild;
} BiTNode, *BiTree;
```



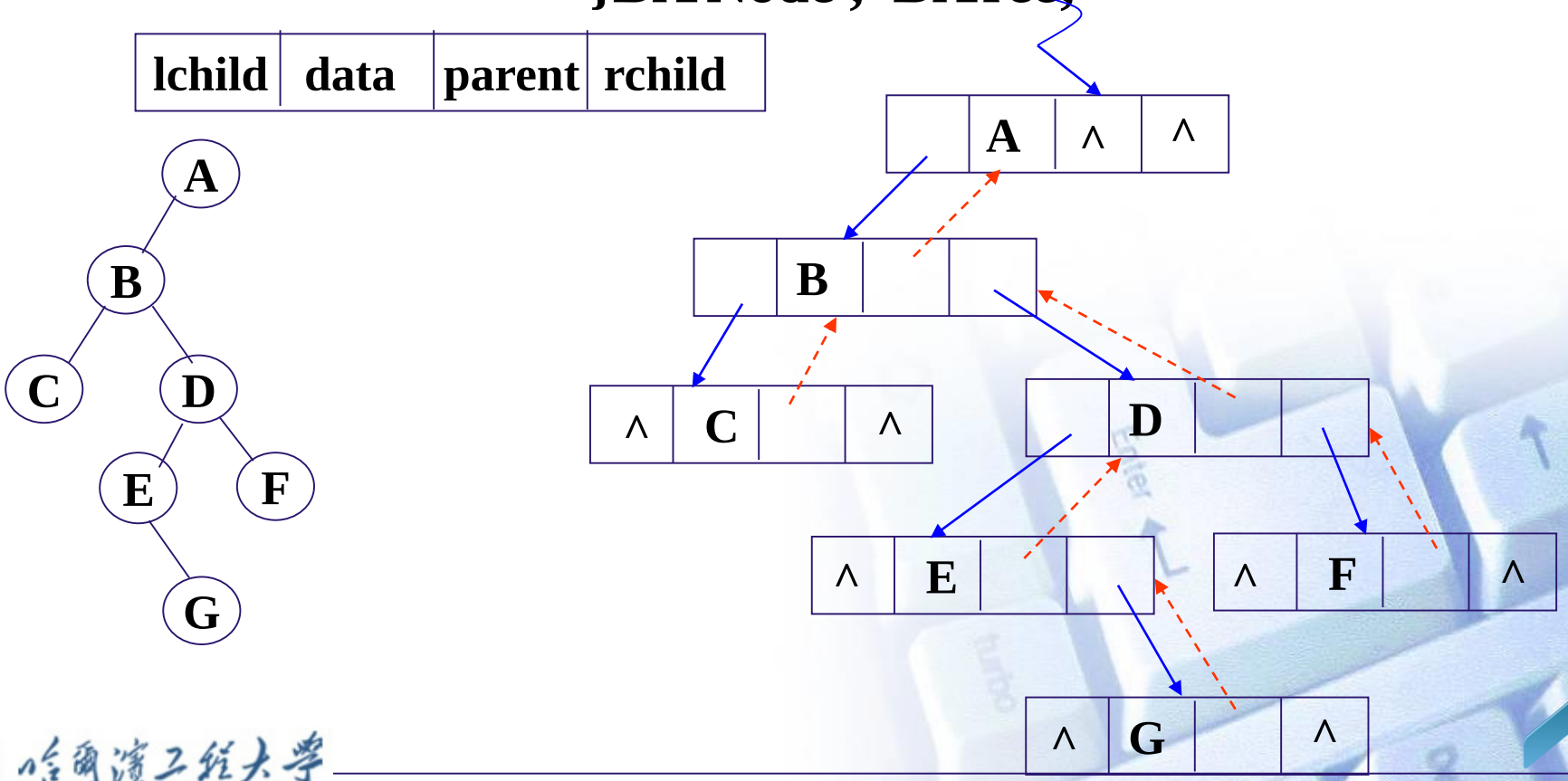


在n个结点的二叉链表中，有n+1个空指针域；2n-(n-1)

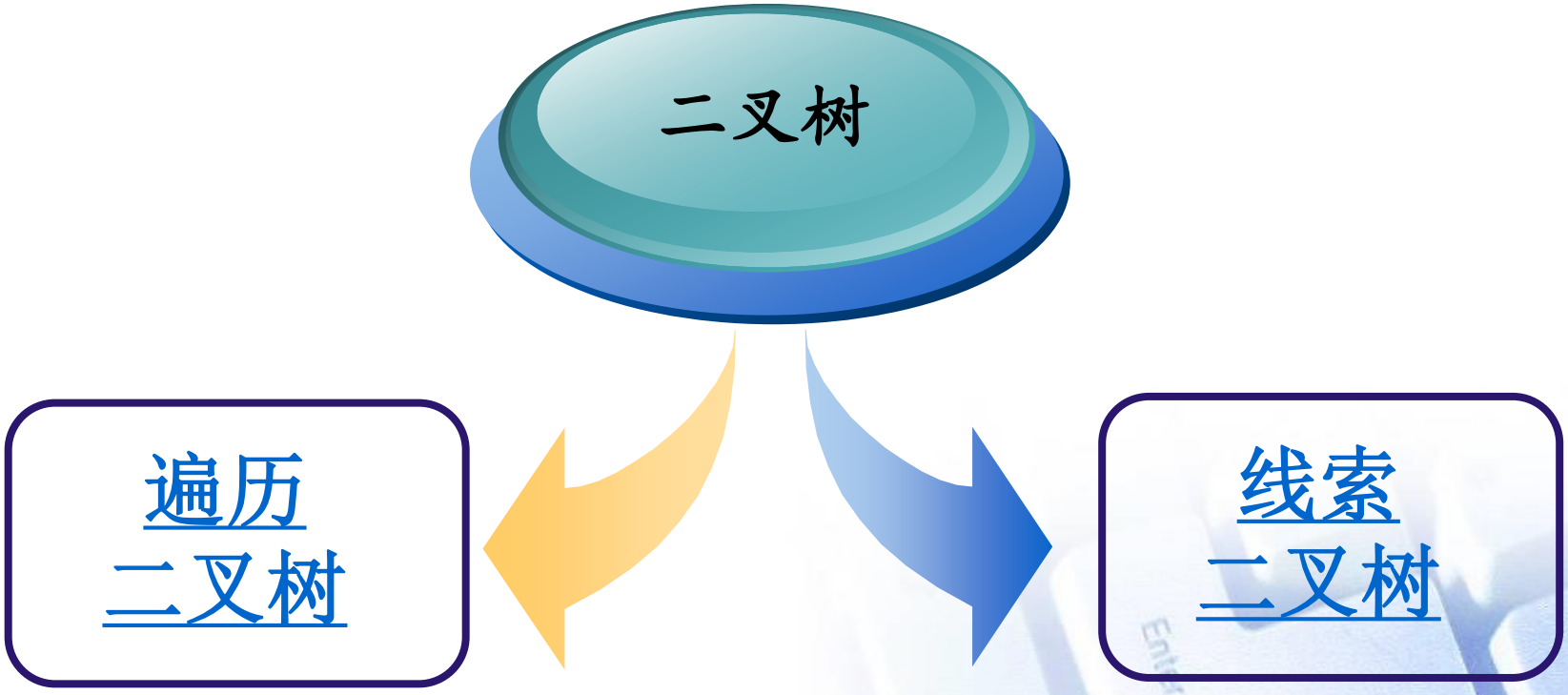
链式存储

三叉链表

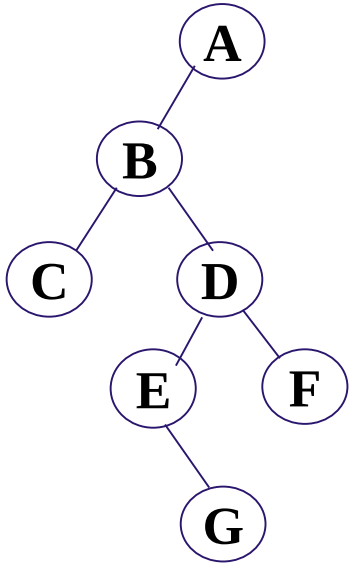
```
typedef struct BiTNode
{TElemType data;
struct BiTNode *lchild, *rchild,*parent;
}BiTNode ,*BiTree;
```



本章内容


二叉树
遍历
二叉树
线索
二叉树

遍　历　顺着某一条搜索路径巡访二叉树中的结点，使得每个结点均被访问一次，而且仅被访问一次



“访问”的含义可以很广，如：输出结点的信息等;同时，遍历和访问是二叉树进行其它操作的基础。

遍 历

- 线性结构而言，每个结点均只有一个后继，只有一条搜索路径
- 二叉树是非线性结构，每个结点有两个后继，而且具有层次结构的特点，则存在如何遍历即按什么样的搜索路径进行遍历的问题？

遍历的过程就是把非线性结构的二叉树中的结点排成一个线性序列的过程。

可以有三条搜索路径：

（1）先上后下的按层次遍历

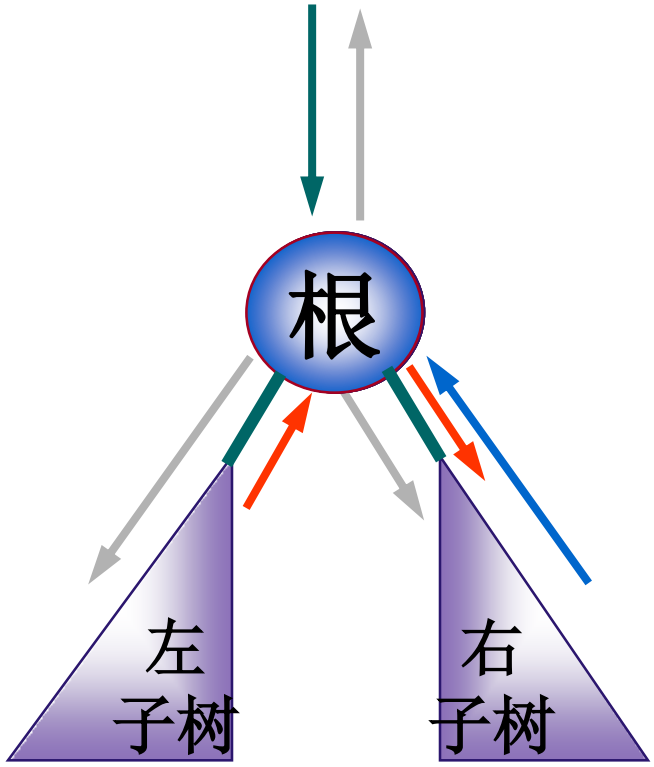
（2）先左（子树）后右（子树）的遍历；

（3）先右（子树）后左（子树）的遍历。

通常，更多的采用先左后右的算法！

- 先左后右的遍历算法，以根为标尺

  访问根结点、遍历左子树、遍历右子树



先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法

- 先（根）序的遍历算法

  若二叉树为空树，则空操作；否则

- 中（根）序的遍历算法

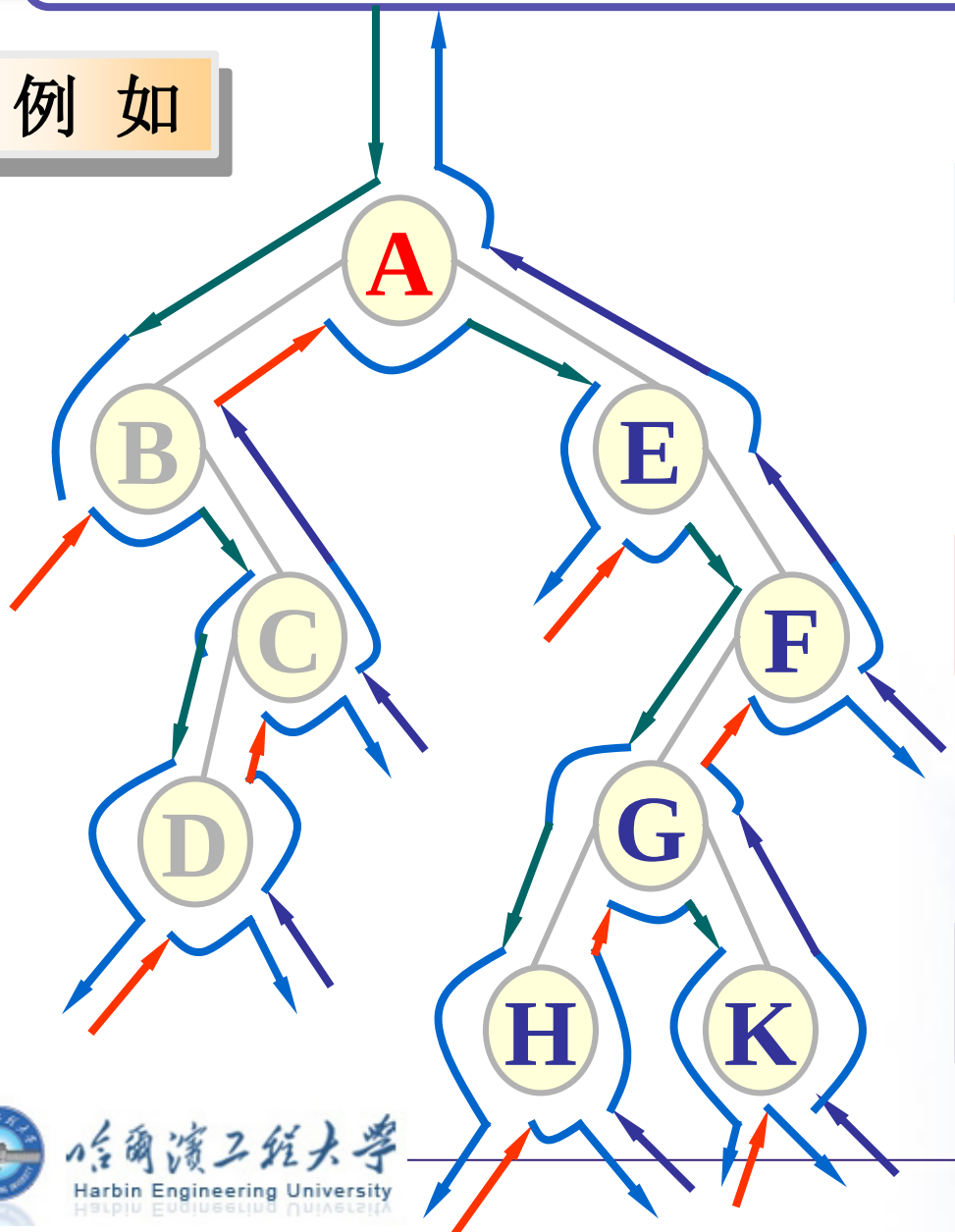
若二叉树为空树，则空操作；否则

- 后（根）序的遍历算法

若二叉树为空树，则空操作；否则

遍历二叉树　　先左后右的遍历算法

例 如



先序序列：

A B C D E F G H K

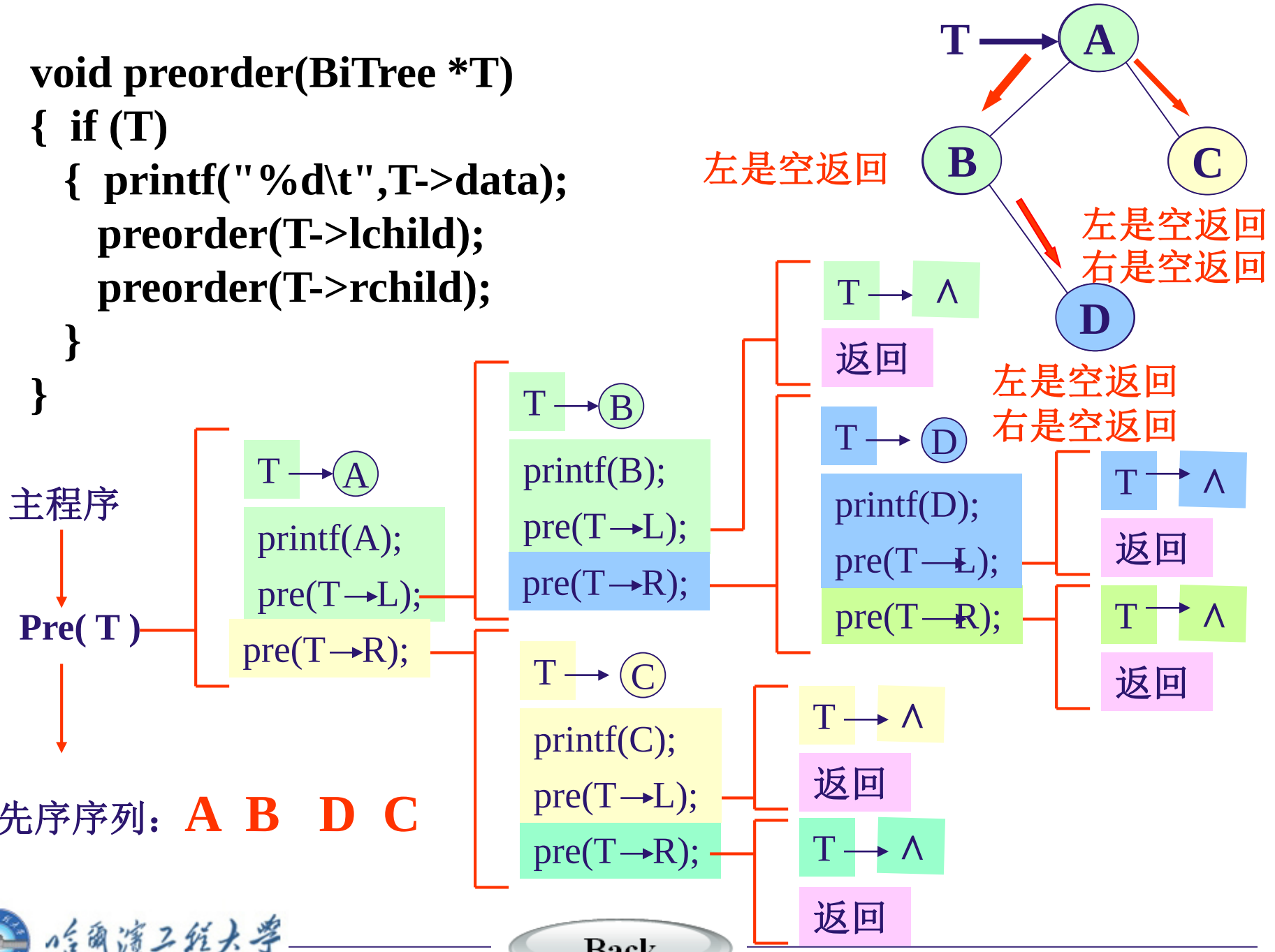
中序序列：

B D C A E H G K F

后序序列：

D C B H K G F E A

```
void preorder(BiTree *T)
{  if (T)
   {  printf("%d\t",T->data);
      preorder(T->lchild);
      preorder(T->rchild);
   }
}
```



递归： 是指函数在执行的过程中调用到自身来完成需要的功能，用递归能解决的问题通常能将问题不断缩小为性质相同但规模更小的问题（递归情况），直到问题足够小能够直接解决（基本情况）。递归算法是一种直接或者间接地调用自身算法的过程。

递归算法一般用于解决三类问题：

(1)数据的概念是按递归定义的

(2)问题解法按递归算法实现。这类问题虽则本身没有明显的递归结构，但用递归求解比迭代求解更简单

(3)数据的结构形式是按递归定义的。如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述。

递归算法解决问题的特点：

(1) 递归就是在过程或函数里调用自身。规则已经确定，同时，相同问题，规模依次变小，传递参数；

(2) 在使用递归策略时，必须有一个明确的递归结束条件，称为递归出口，否则将无限进行下去（死锁）。

(3) 递归算法解题通常显得很简洁，但递归算法解题的运行效率较低。

递归算法一般有三个要求：

（1）每次调用在规模上都有所缩小(通常是减半)；

（2）相邻两次重复之间有紧密的联系，前一次要为后一次做准备(通常前一次的输出就作为后一次的输入)；

（3）在问题的规模极小时必须用直接给出解答而不再进行递归调用，因而每次递归调用都是有条件的(以规模未达到直接解答的大小为条件)，无条件递归调用将会成为死循环而不能正常结束。

递归函数的步骤：

①写出迭代公式；②确定递归终止条件；③将①②翻译成代码。

例子，求：f（n）=1+2+3+……+n的值

①写出迭代公式：迭代公式为 Sum(n-1)= (Sum(n-1))+n;

②确定递归终止条件：f（1）=1就是递归终止条件

③将①②翻译成代码：将迭代公式等号右边的式子写入return语句中，即return （Sum（n-1））+n; 判断语句：if（n==1） return 1;

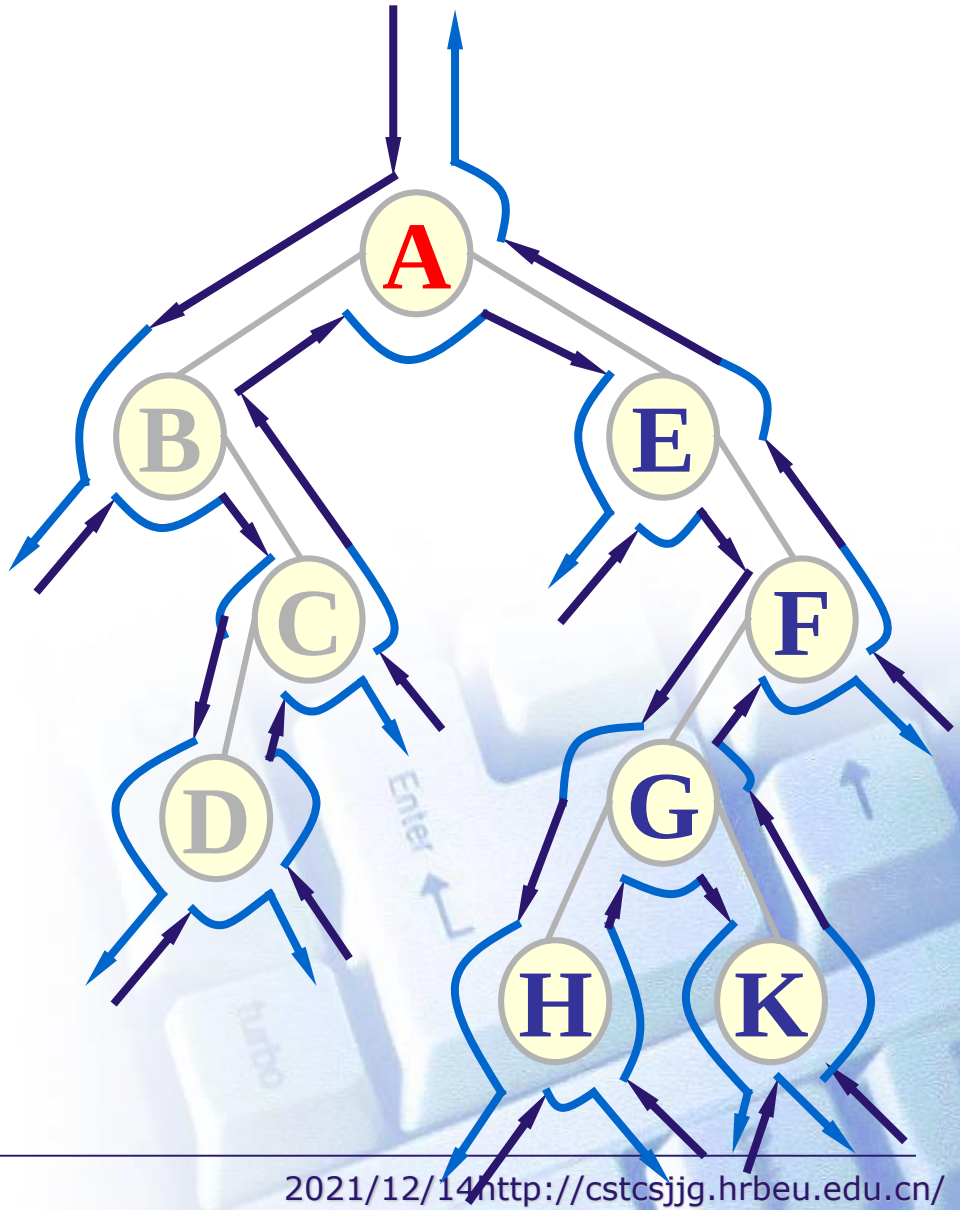
递归代码:

```
long Sum ( int n)
{
    if(n==1)   return 1;
    return  (Sum(n-1))+n;
}
```

- 三种遍历算法的比较

相同点

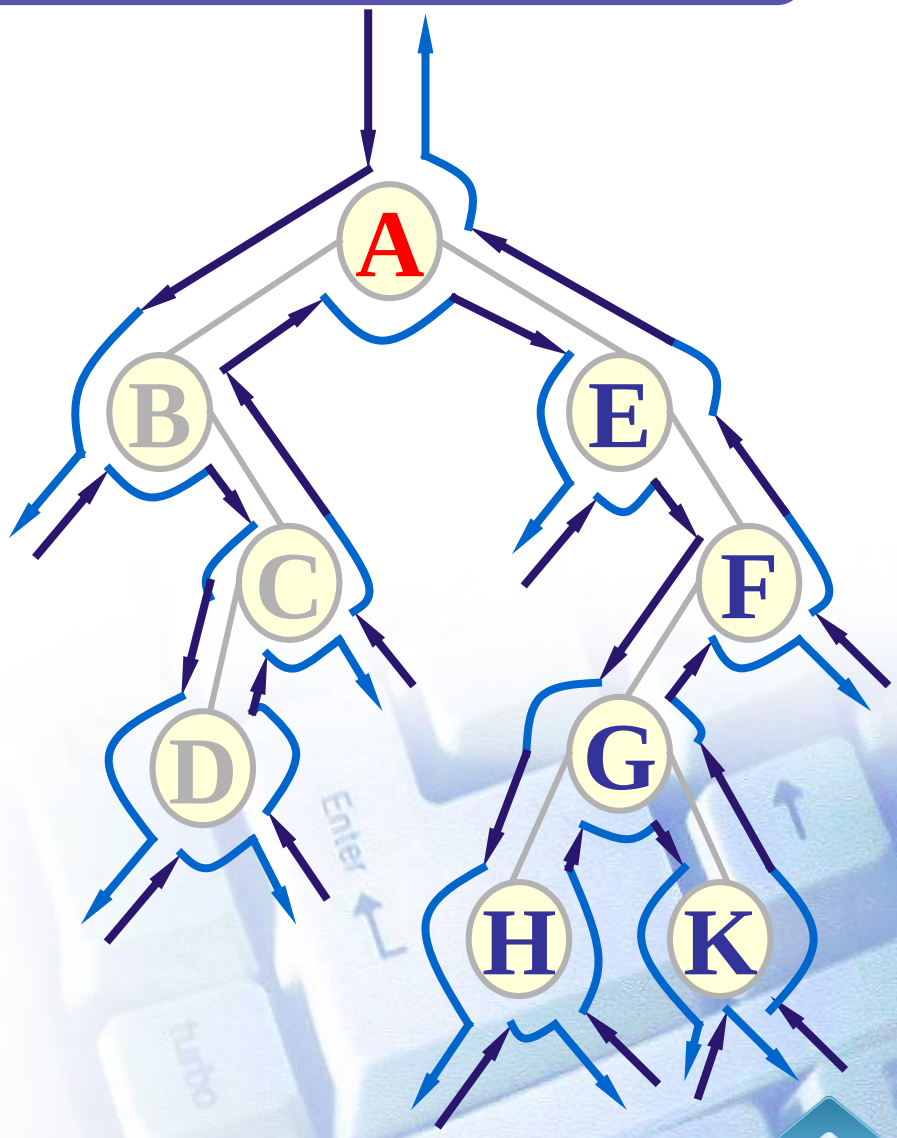
如果把访问根结点这个不涉及递归的语句抛开，则三个递归算法走过的路线是一样的



- 三种遍历算法的比较

不同点

- 前序遍历是每进入一层递归调用时先访问根结点，然后再依次向它的左、右子树执行递归调用
- 中序遍历是在执行完左子树递归调用后再访问根结点，然后向它的右子树递归调用
- 后序遍历则是在从右子树递归调用退出后访问根结点



- 递归算法

优点

形式简洁，可读性好

正确性容易得到证明

可以给程序的编制和调试带来很大的方便

缺点

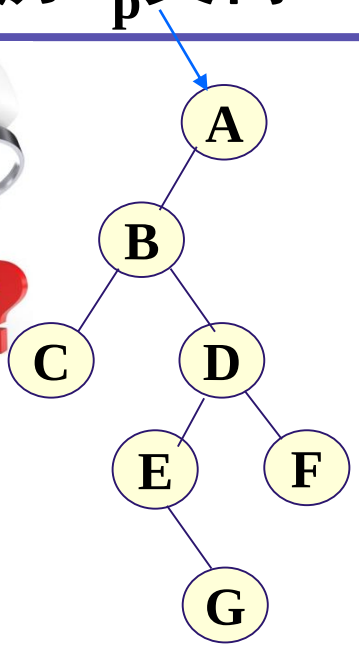
比非递归调用消耗的时间与存储空间多，运行的效率较低

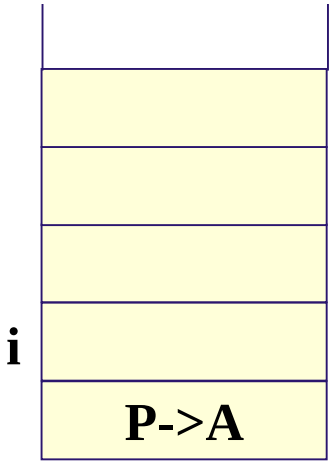
递归算法转化为非递归算法

- 所有的递归算法都可转化成相应的非递归算法
- 将递归算法改成相应的非递归算法需要一个栈来记录调用返回的路径
- 通过分析递归调用的执行过程，并观察栈的变化就可得到相应的非递归算法

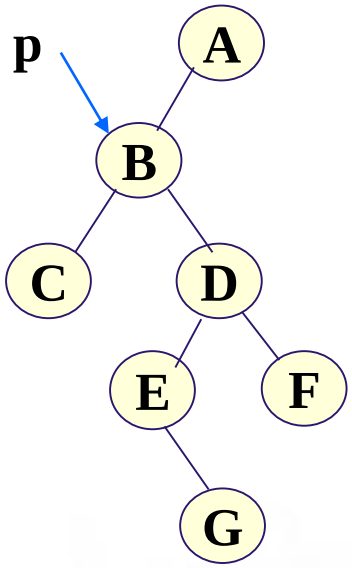
遍历二叉树 算法的非递归描述

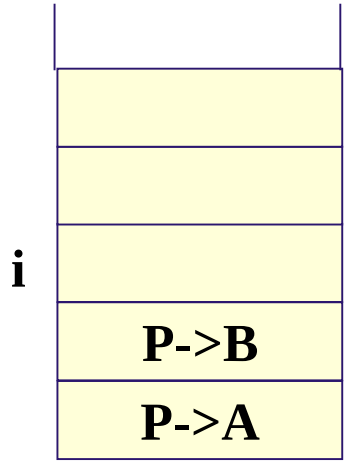
中序遍历



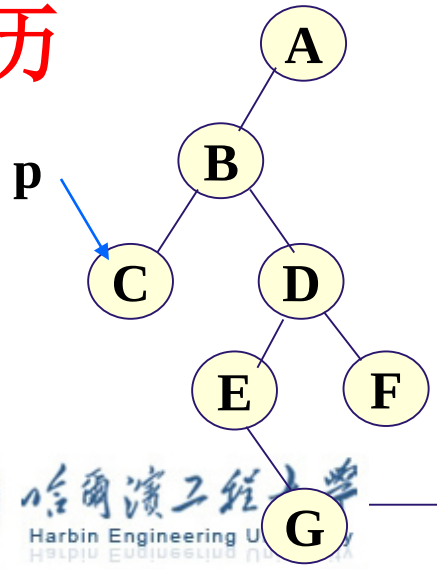


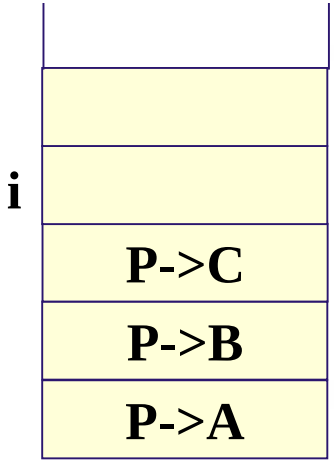
(1)



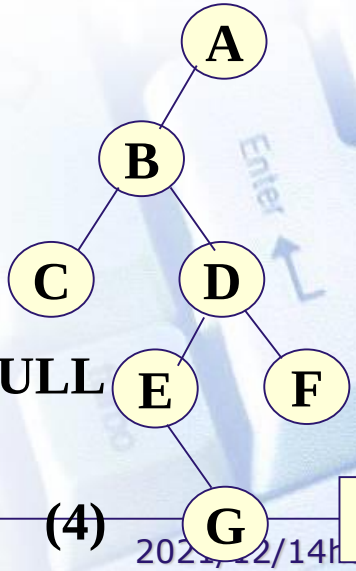


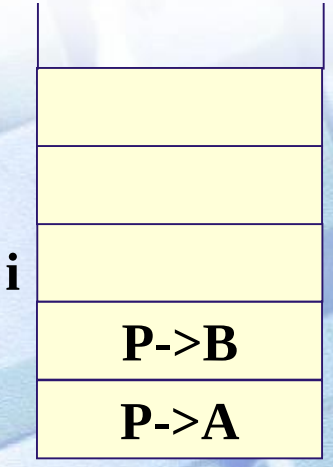
(2)





(3)

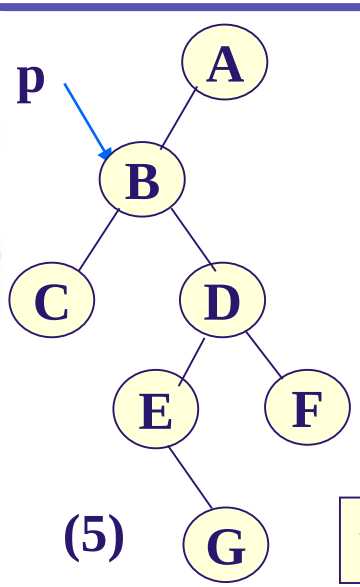


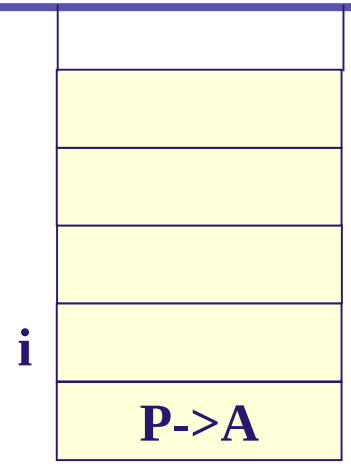


(4)

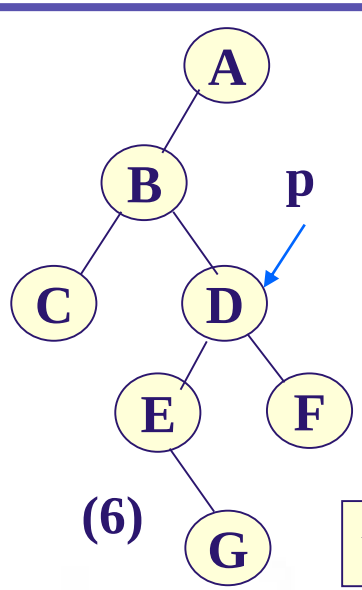
访问：C

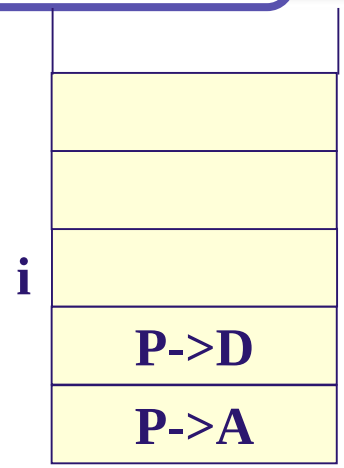
遍历二叉树　　算法的非递归描述



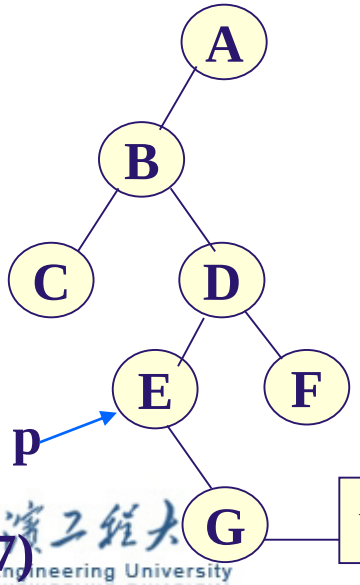


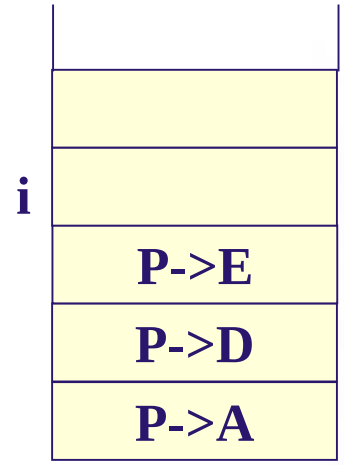
(5)　访问：C B



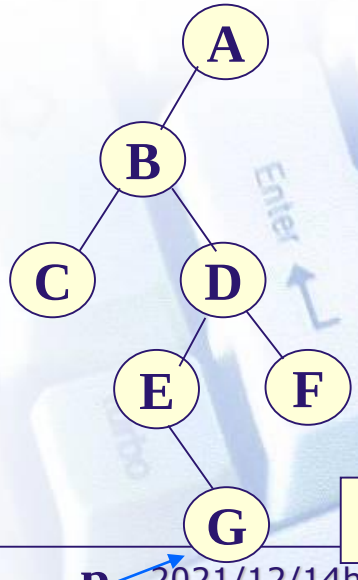


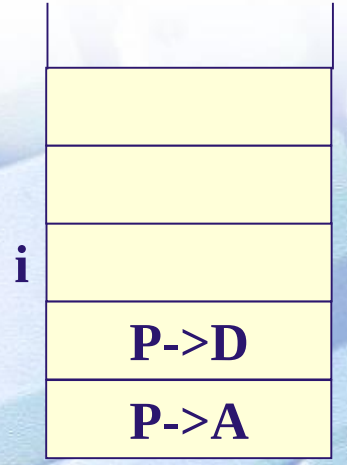
(6)　访问：C B





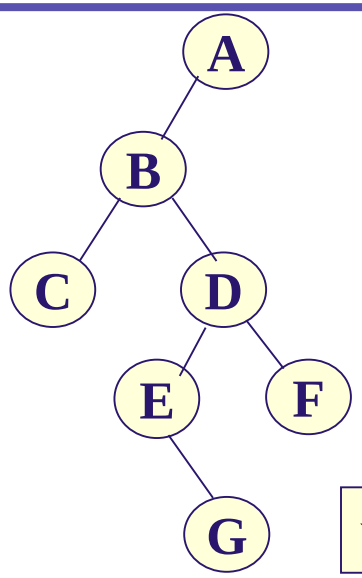
(7)　访问：C B

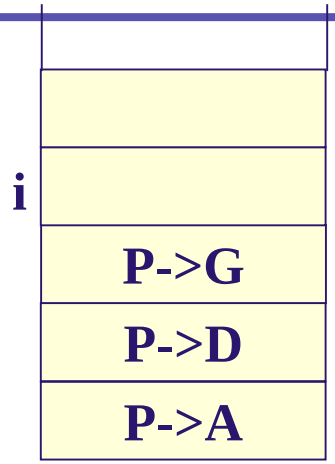




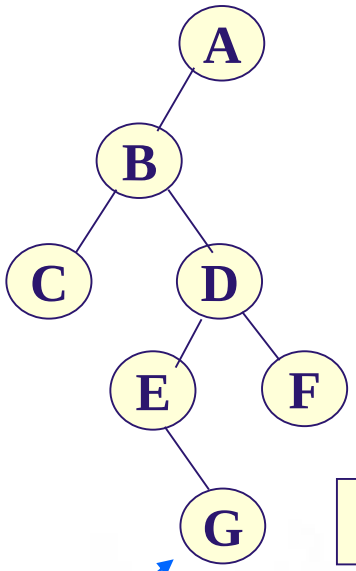
(8)　访问：C B E

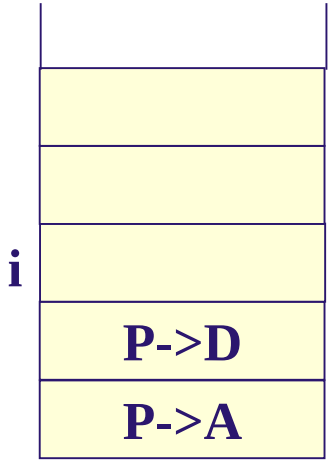
遍历二叉树　　算法的非递归描述





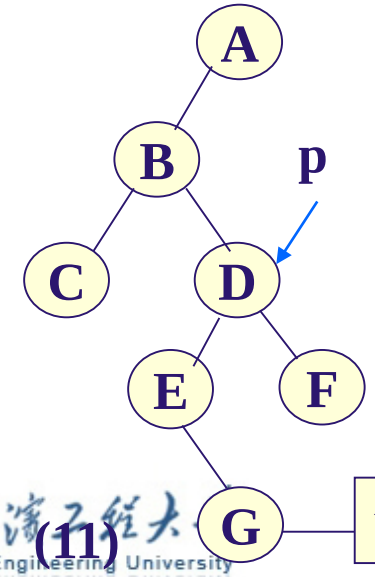
访问：C B E (9)

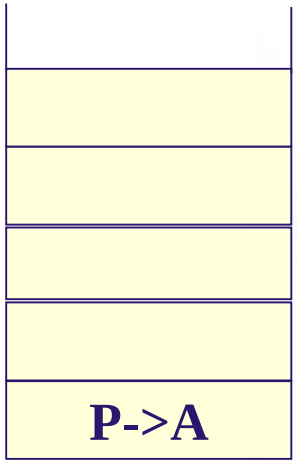




访问：C B E G (10)

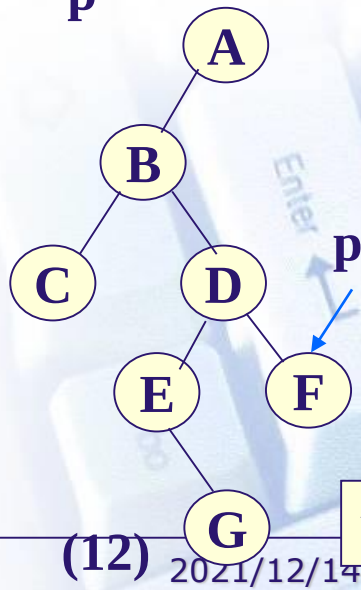
p=NULL

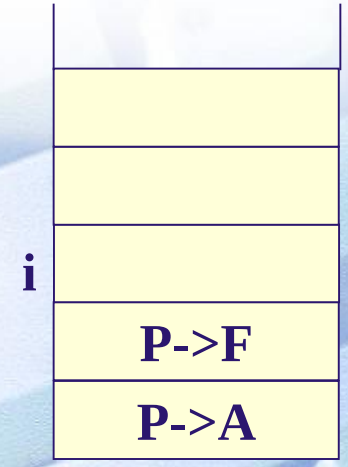




访问：C B E G D (11)

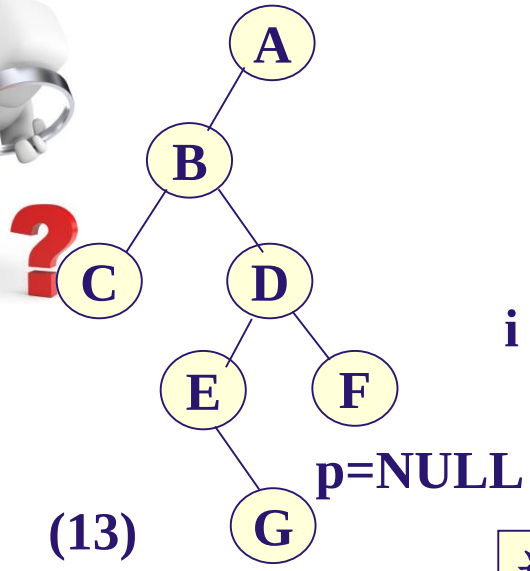
p

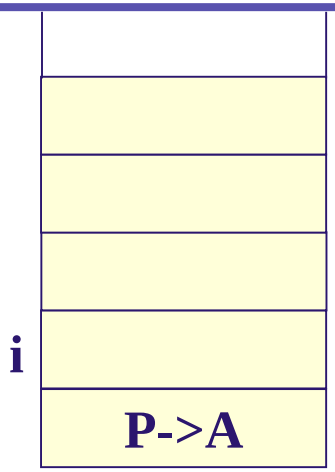




访问：C B E G D (12)

遍历二叉树　　算法的非递归描述

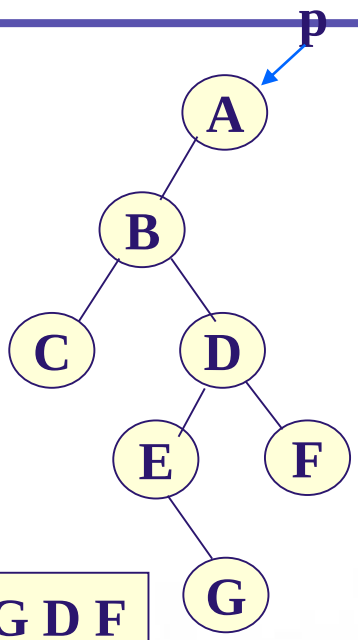


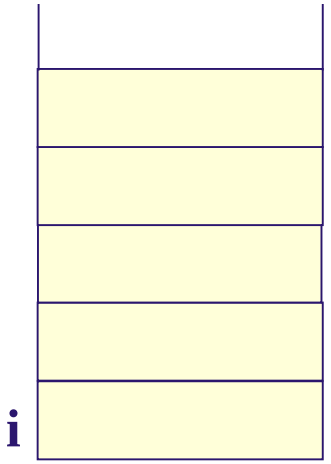


p=NULL

(13)

访问：C B E G D F

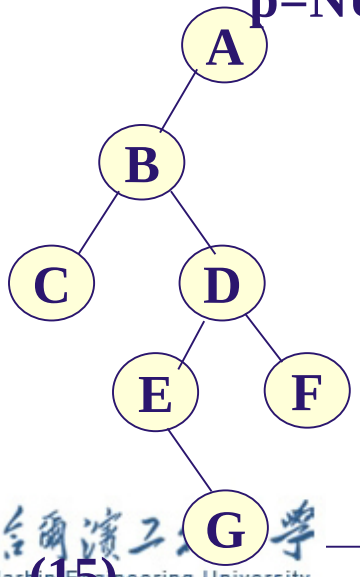


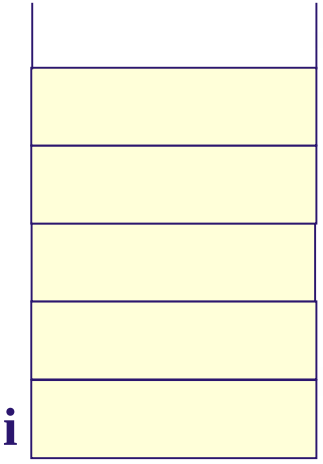


(14)

访问：C B E G D F A

p=NULL





(15)

访问：C B E G D F A

- 递归算法转化为非递归算法

  - 将递归算法改成相应的非递归算法需要一个栈来记录调用返回的路径

算法非递归描述

算法思想

二叉树中序遍历的非递归算法

- 设置堆栈S，根指针进栈
- 栈S不空，重复下面两步，直到栈空为止
- 栈顶指针不空：遍历左子树，即向左走到尽头（即空指针入栈）
- 栈顶指针为空：退到上一层（空指针退栈）；若从左子树返回，访问当前层根结点；向右进一步（遍历右子树）

```
void inorder(BiTree T, Status(*Visit)(TElemType e))
  { //采用二叉链表存储结构，中序遍历、visit是对数据元素操作的应用函数
    InitStack(S);  p=T;
    while (p || !StackEmpty(S))
       {
        if (p)
           {
                Push(S,p);    p=p->lchild;
           } //根指针进栈，遍历左子树
         else
           {
              Pop(S,p);  //根指针出栈
              if (!Visit(p->data)) return ERROR;
              p=p->rchild;//遍历右子树
           }
       }
     return OK;
  } //InOrderTraverse
```

- 二叉树表示表达式
- 统计二叉树中叶子结点的个数
- 求二叉树的深度
- 建立二叉树的存储结构
- 二叉树遍历的重要性质
- 在二叉树上查询某个结点
- 输出后序序列的逆序

二叉树表示表达式

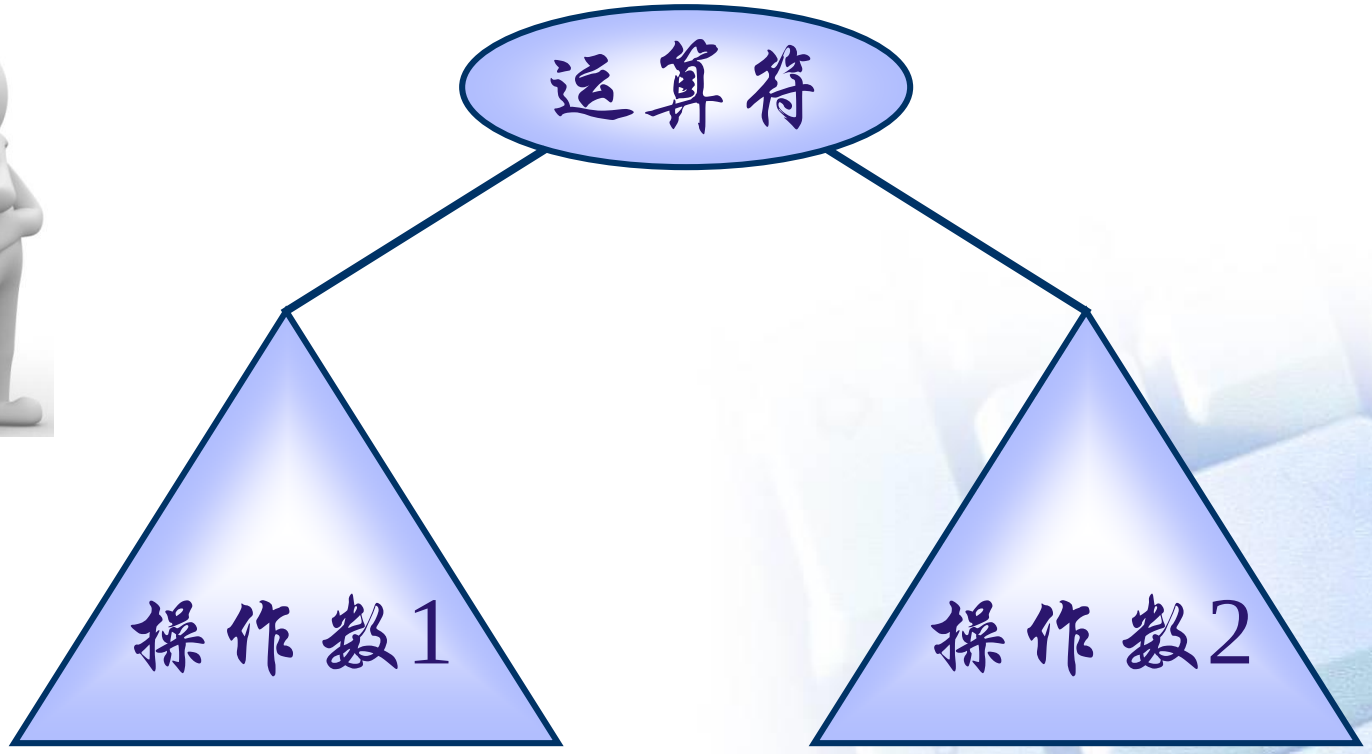
基本思想

- 数或简单变量（条件）
- 其他

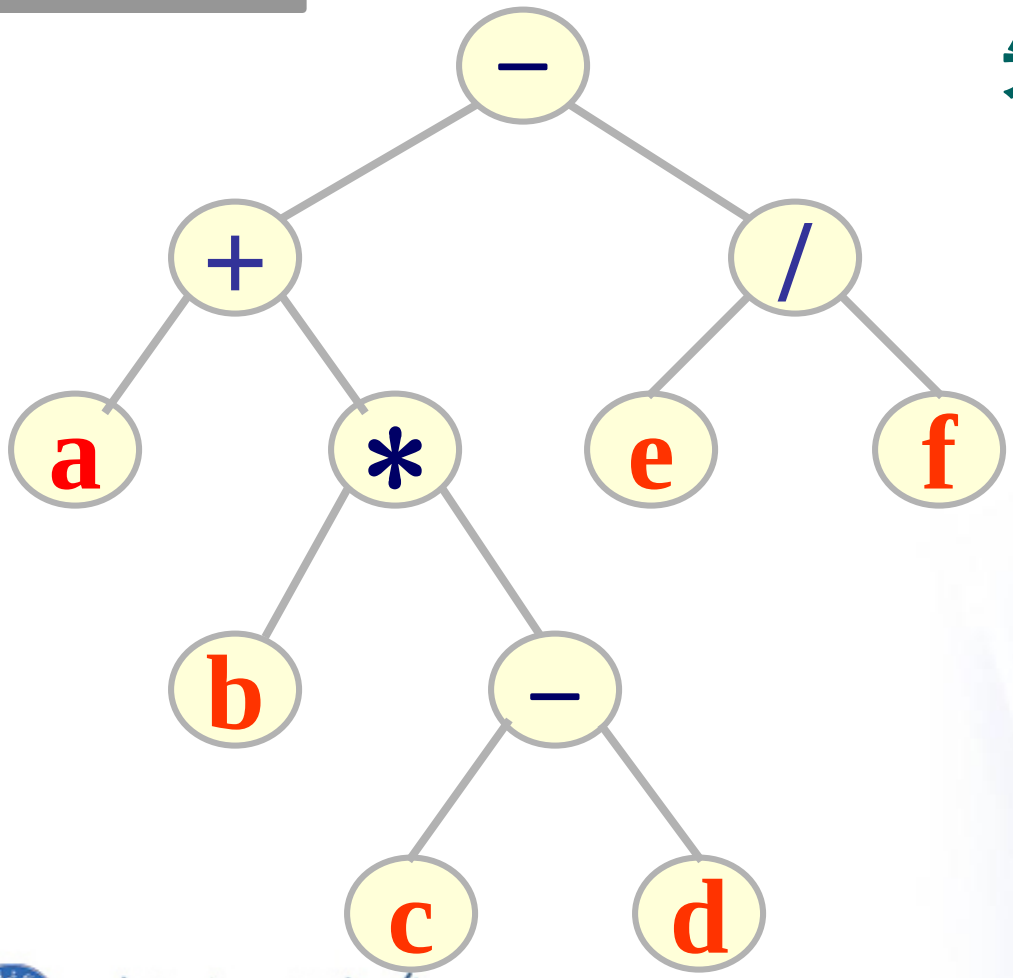
  <第一操作数>（运算符）<第二操作数>

  - ✓运算符——根
  - ✓第一操作数——左子树
  - ✓第二操作数——右子树
- 操作数本身可以是表达式

表达式 = (操作数1)(运算符)(操作数2)



例　如　a+b*(c-d)-e/f 的二叉树表示



先序（前缀表达式）

-+a*b-cd/ef

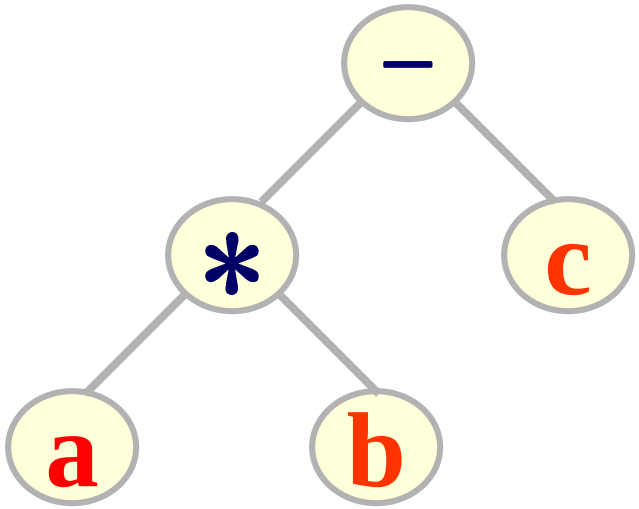
中序（中缀表达式）

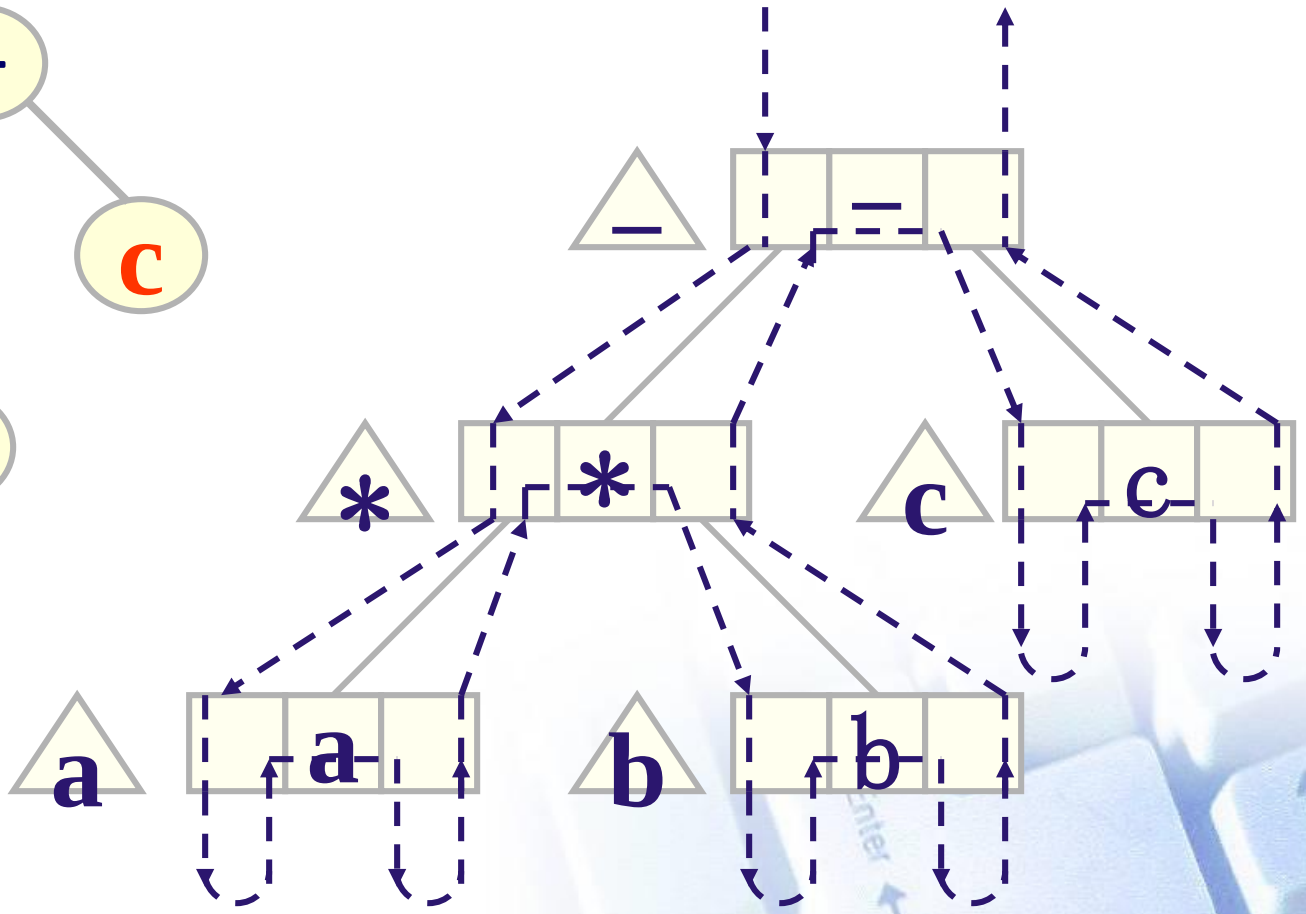
a+b*c-d-e/f

后序（后缀表达式）

abcd-*+ef/-

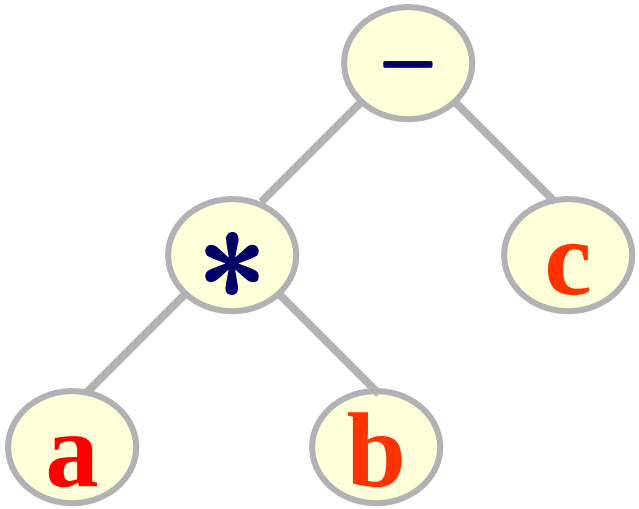
遍历二叉树　　遍历算法的应用举例

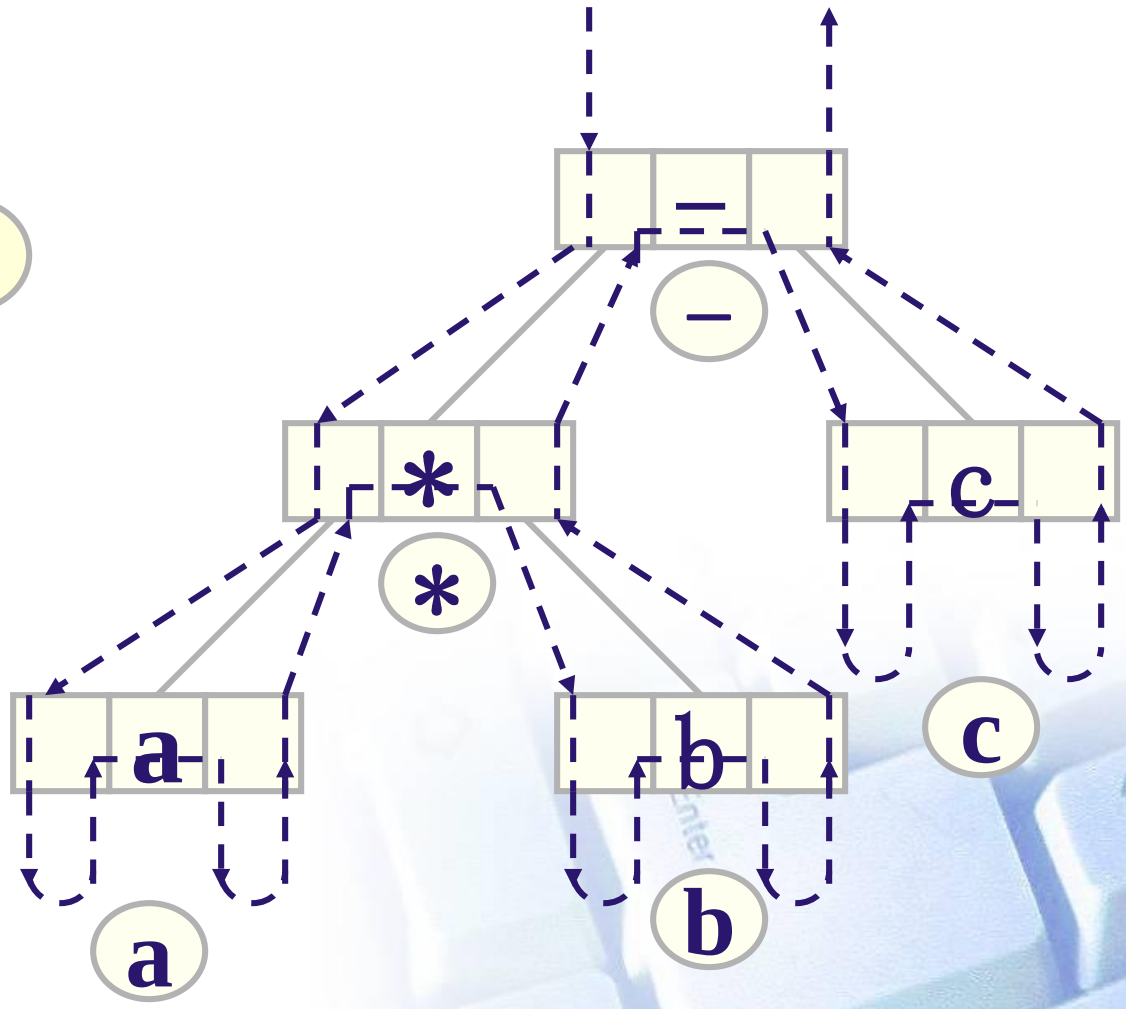




先序遍历：- * a b c（波兰式）

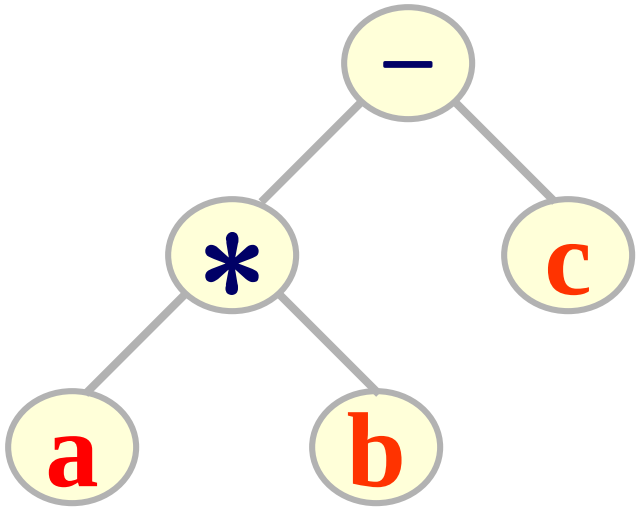
遍历二叉树　　遍历算法的应用举例

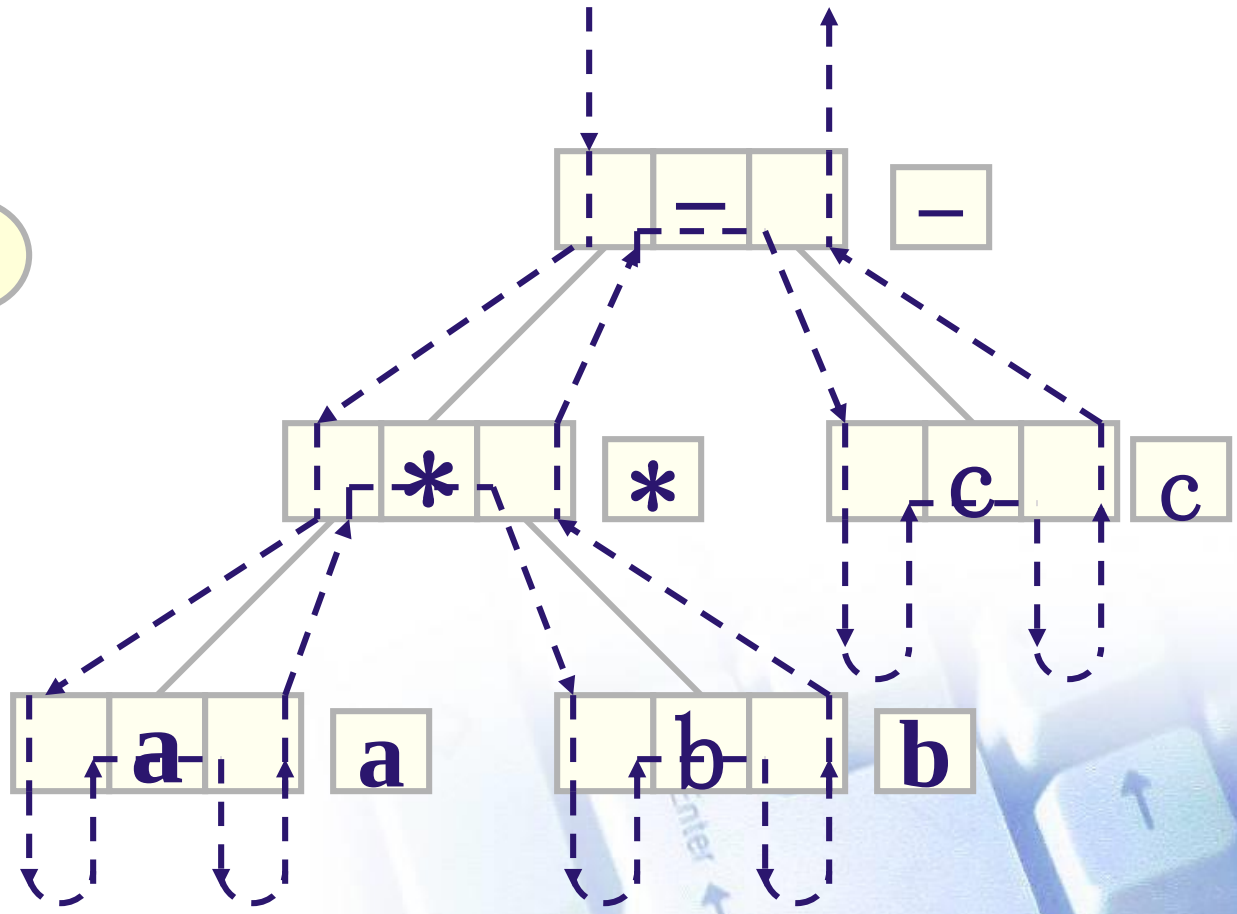




中序遍历：　a * b - c

遍历二叉树　　遍历算法的应用举例





后序遍历：　a b * c –　（逆波兰式）

统计二叉树中叶子结点的个数

基本思想

- 先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数
- 即，需在遍历算法中增添一个“计数”的参数，并将算法中“访问结点”的操作改为:若是叶子，则计数器增1。

Click

统计二叉树中叶子结点的个数

```
void CountLeaf (BiTree T,  int &count)
 {
    if ( T )
     {
        if ((!T->lchild)&& (!T->rchild))
          count++; // 对叶子结点计数
        CountLeaf( T->lchild, count);
        CountLeaf( T->rchild, count);
     } // if
 } // CountLeaf
```

- 注意：调用本函数之前预置实参count为0

- 统计叶子结点个数的另一种方法（后序遍历）

基本思想

- 若二叉树为空，则叶子结点的个数为零
- 若二叉树中只有一个(根)结点，则叶子结点的个数为1
- 否则，整个二叉树中的叶子结点个数等于其左子树中的叶子结点个数和其右子树中的叶子结点个数之和

Click

后序遍历统计二叉树中叶子结点的个数

```
int CountLeaf (BiTree T)
 {// 返回指针T所指二叉树中所有叶子结点个数
    if (!T )    return 0;
      if (!T->lchild && !T->rchild)    return 1;
     else
     {   m = CountLeaf( T->lchild);
         n = CountLeaf( T->rchild);
        return (m+n);
     } //else
 } // CountLeaf
```

求二叉树的深度（后序遍历）

基本思想

- 首先分析二叉树的深度和它的左、右子树深度之间的关系
- 若二叉树为空树，则深度为0
- 否则，二叉树的深度应为其左、右子树深度的最大值加1
- 由此，需先分别求得左、右子树的深度

```
int Depth (BiTree T ){ // 返回二叉树的深度
   if ( !T )    depthval = 0;
   else   {
     depthLeft = Depth( T->lchild );
     depthRight= Depth( T->rchild );
     depthval = 1 + (depthLeft > depthRight  ?
                          depthLeft : depthRight);
   }
   return depthval;
```

求二叉树的深度（另一种角度分析）

基本思想

- 首先分析二叉树的深度和结点的“层次”间的关系
- 从二叉树深度的定义还可知，二叉树的深度即为其叶子结点所在层次的最大值。由此，可通过遍历求得二叉树中所有结点的“层次”，从中取其最大值
- 算法中需引入一个计结点层次的参数 **dval**

Click

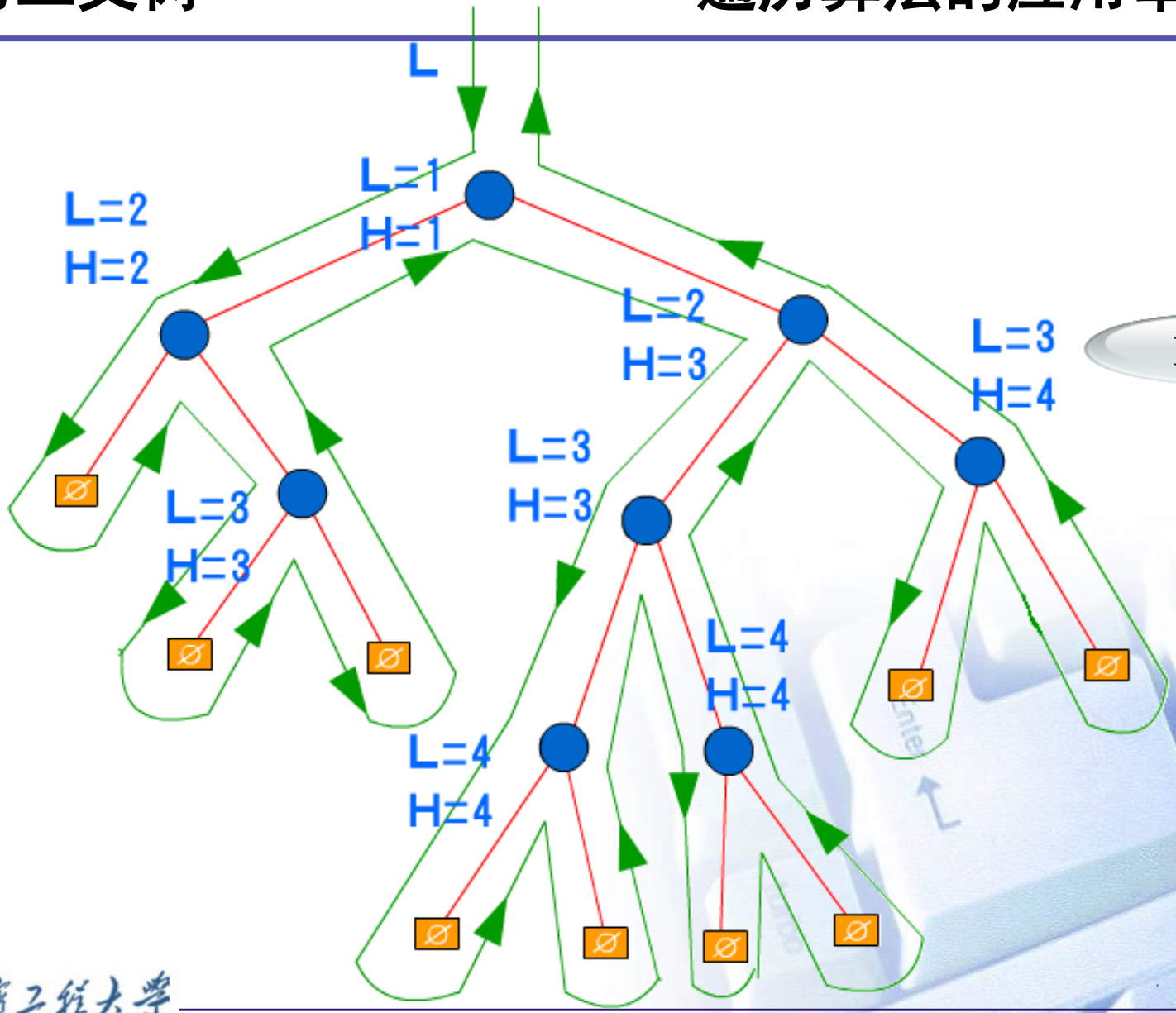
```
void Depth(BiTree T , int level, int &hval){
  if ( T ) {
      if (level>hval)  hval = level;
      Depth( T->lchild, level+1, hval );
      Depth( T->rchild, level+1, hval );
  }
}
```

Click

// 调用之前层次数level 的初值为 1

//深度hval 的初值为 0

遍历二叉树　　遍历算法的应用举例



Back

- 建立二叉树的存储结构

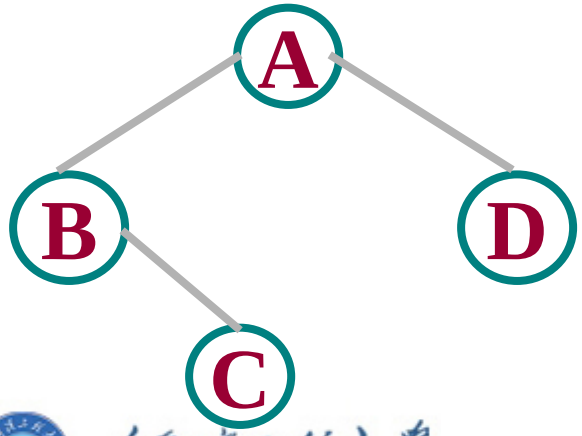
例　如

以字符串形式“根 左子树 右子树”定义

空树　以空白字符“■”或Φ表示

只含一个根结点的二叉树

以字符串“A■■”表示



以下列字符串表示

A(B(■,C(■,■)),D(■,■))

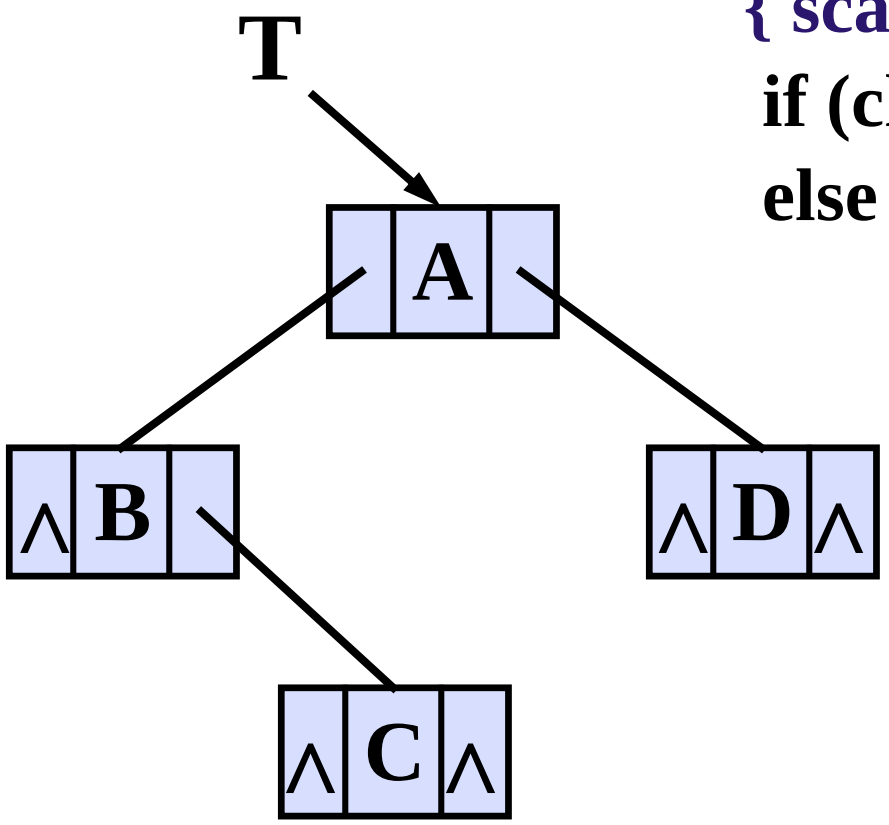
建立二叉树的存储结构

算法思想

- 读一字符ch
- 如果ch为Φ，T=NULL
- 否则
  - 申请结点
  - 生成结点T
  - 构造T的左子树
  - 构造T的右子树
- 返回

```
void CreateBiTree(BiTree &T) {
    scanf(&ch);
    if (ch==' ') T = NULL;
    else {
       if(!( T =(BiTNode*) malloc(sizeof(BiTNode))))
            exit (OVERFLOW)
            T->data = ch;                  // 生成根结点
            CreateBiTree(T->lchild);   // 构造左子树
            CreateBiTree(T->rchild);   // 构造右子树
           }
} // CreateBiTree
```

A B ■ C ■ ■ D ■ ■




```
void CreateBiTree(BiTree &T)
{ scanf(&ch);
 if (ch==' ') T = NULL;
 else {
        T = (BiTNode*)
         malloc(sizeof( BiTNode));
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
       }
}
```

- 二叉树遍历的重要性质

例　如　仅知二叉树的先序序列“abcdefg”不能唯一确定一棵二叉树，如果同时已知二叉树的中序序列“cbdaegf”，则会如何？

二叉树的先序序列　根　左子树　右子树

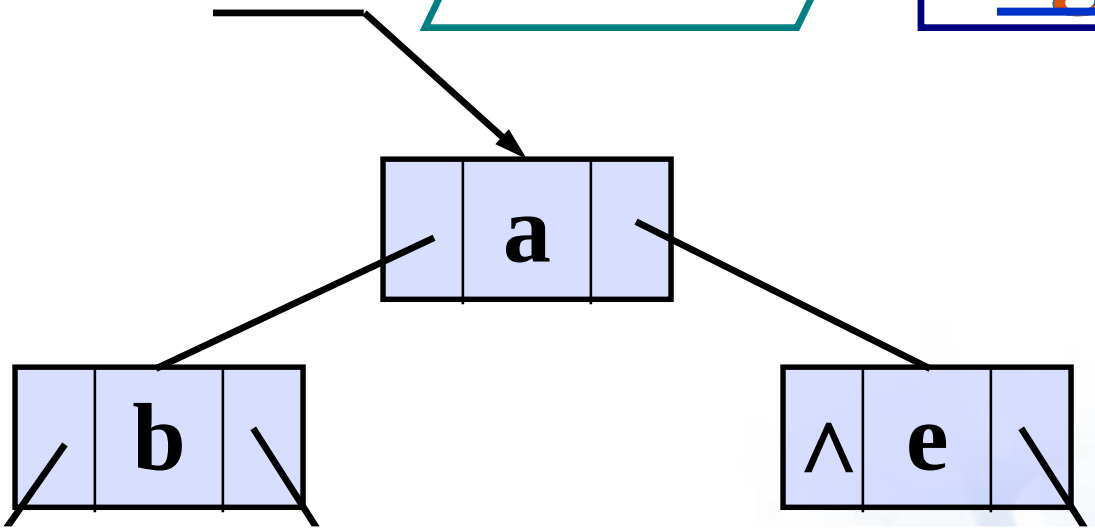
二叉树的中序序列　左子树　根　右子树

二叉树的后序序列　左子树　右子树　根

例　如

a b c d e f g　先序序列

c b d a e g f　中序序列



思考：先序、中序、后序序列中任意给定两个序列就可以画出该二叉树吗？为什么？

推 论

二叉树遍历的重要性质：

- 若已知二叉树的先序序列和中序序列可唯一确定一棵二叉树；
- 若已知二叉树的后序序列和中序序列可唯一确定一棵二叉树。

- 在二叉树上查询某个结点

问　题

假设给定一个和二叉树中数据元素有相同类型的值，在已知二叉树中进行查找，若存在和给定值相同的数据元素，则返回函数值为 **TRUE**，并用引用参数返回指向该结点的指针；否则返回函数值为 **FALSE**

算法思想

- 若二叉树为空树，则二叉树上不存在这个结点，返回 FALSE；
- 否则，和根结点的元素进行比较

  若相等，则找到，即刻返回指向该结点的指针和函数值 TRUE，从而查找过程结束

  否则，在左子树中进行查找（递归）

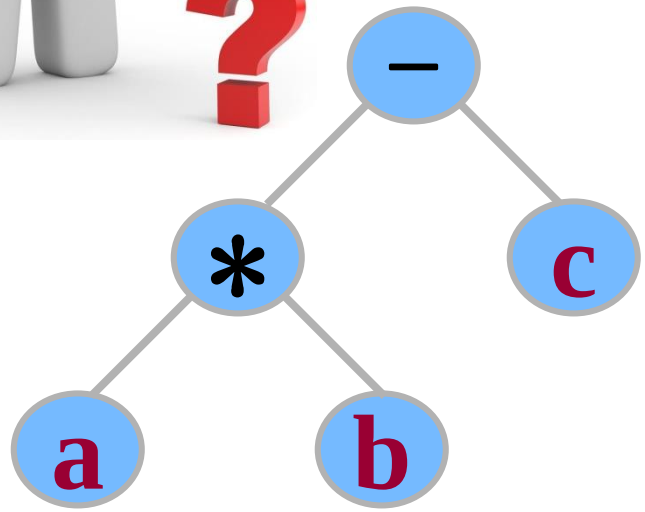
  若找到，则返回 TRUE

  否则，返回在右子树中进行查找的结果

因右子树上查找的结果即为整个查找过程的结果，即若找到，返回的函数值为 TRUE，并且已经得到指向该结点的指针，否则返回的函数值为 FALSE

```
bool Locate (BiTree T, ElemType x, BiTree &p)
{// 存在和x相同的元素，则p指向该结点并返回 TRUE
   否则p= NULL 且返回 FALSE
 if (!T){ p = NULL;      // 空树中不存在该结点
          return FALSE; }
 else{
     if (T->data==x){    // 找到所求结点
         p = T; return TRUE; }    //else
     if (Locate(T->Lchild,x,p))
         return TRUE;  // 在左子树中找到该结点
       else return(Locate(T->Rchild,x,p));
                  // 返回在右子树中查找的结果
    } // else
}// Locate
```

- 输出后序序列的逆序



- 观察一下后序遍历的过程，发现规律
- 先访问根，先序遍历右子树，先序遍历左子树，即为后序遍历的逆序

后序序列　**ab*c-**

后序序列的逆序　**-c*ba**

```
void preorder(tree T){
   //输出后序序列的逆序
   if (T){
        printf("%d", T->data);
        preorder(T->rchild);
        preorder(T->lchild);
   }
}
```

- 交换二叉树中所有结点的左右子树

```
void exchg_tree(BiTre *T)
{   //采用后序遍历方法，交换每一个结点的左右子树
    if (BT){ //非空
        exchg_tree(BT->lchild); //交换左子树所有结点指针
        exchg_tree(BT->rchild); //交换右子树所有结点指针
        p=BT->lchild;          //交换根结点左右指针
        BT->lchild=BT->rchild; BT->rchild =p;
     }
}
```

- 二叉树中，请设计算法统计元素值大于e的结点的个数。（13年）

```
int Count_e (BiTree T, int &count,  int  e )
    //递归求以孩子左右子树中的个数
{   if (T)
    {       if (T->data > e)   count++;
            Count_e (t->lchild, count, e);  //左子树递归
            Count_e (t->rchild, count, e);  //右子树递归
    }
}//结束count_e
```

- 评分标准：递归的出口和判定书写正确给5分，递归正确给7分，统计正确返回给3分，错误酌情扣分。

- 设计算法，统计一棵二叉树中所有非叶结点的数目及其深度。（11年）

```
void Count(BiTree bt, int level，  &n, &depthval)
//统计二叉树bt上非叶子结点数n、二叉树深度为depthval
{    if (bt)
   {   if (bt->lchild！=null || bt->rchild！=null)    n++；
       // 非叶子结点
     if (level>depthval)  depthval = level;  //统计深度
     Count(bt->lchild, level+1, n，  depthval);
     Count(bt->rchild, level+1, n，  depthval);
   }
}//结束 Count
```

- 评分标准：有递归遍历给5分，统计深度和数目正确各给5分。

- 设计算法返回二叉树T的先序序列的最后一个结点的指针，要求采用非递归形式，且不许用栈。（10年）

```
BiTree LastNode(BiTree bt)
{   BiTree p=bt;
    if(bt==null)  return(null);
   else while(p)
        if (p->rchild) p=p->rchild;
        else if (p->lchild) p=p->lchild;
             else return(p);
}
```

- 评分标准：判断和循环正确给5分，左分支正确给2分，右分支正确给2分结果返回给1分。

本章内容

1 线索二叉树定义

2 线索二叉树的遍历

3 中序遍历线索化二叉树的算法

定 义

二叉树的遍历本质

- 是将一个复杂的非线性结构转换为线性结构，使每个结点都有了唯一前驱和后继
- 遍历序列[illegible]后序序列
- 对于二[illegible]是方便的，其
- 显然，[illegible]程中产生的，如果将这些信息在[illegible]遍历时就保存起来，则在以后再次需要对二叉树进行“遍历”时则可以将二叉树视作线性结构进行访问操作

如何保存在遍历过程中得到的前驱和后继信息？

- 最简单的办法

  在结点中增加两个指针域分别指向该结点的前驱和后继

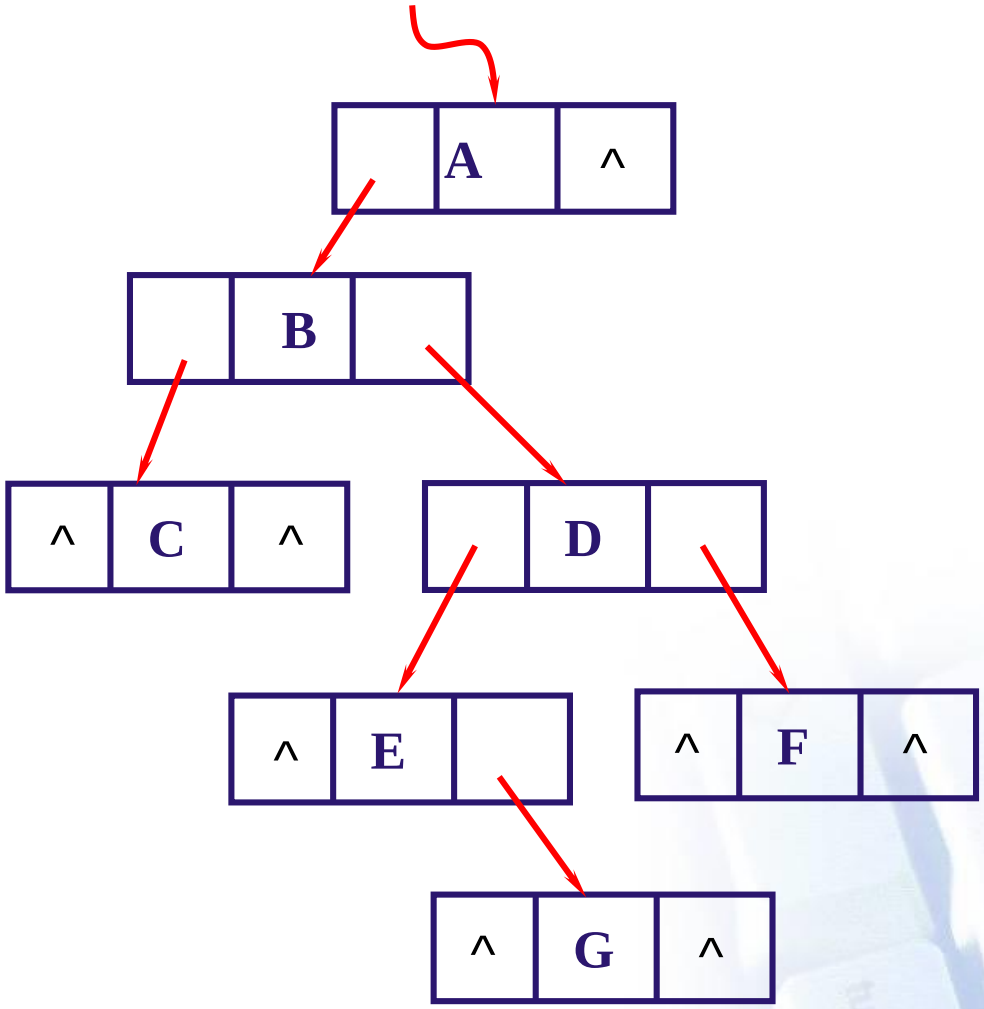
  缺点：存储结构的存储密度大大降低，浪费

- 另一种方法（寻求存储结构中的空指针域）
- 灵感： 一个含 n 个结点的二叉链表中有 n+1 个链域的值为“NULL”
- 思想：利用这些空的指针域存放指向前驱和后继的信息
- 问题：引起混淆

是左孩子还是前驱指针？

是右孩子还是后继指针？

Click


A
^
B
^
C
^
D
^
E
^
F
^
G
^

- 解决办法

利用这些空链域来存放结点的前趋和后继的信息。为避免混淆，需在结点中增加两个标志域

lchild	ltag	data	rtag	rchild

(特征位)

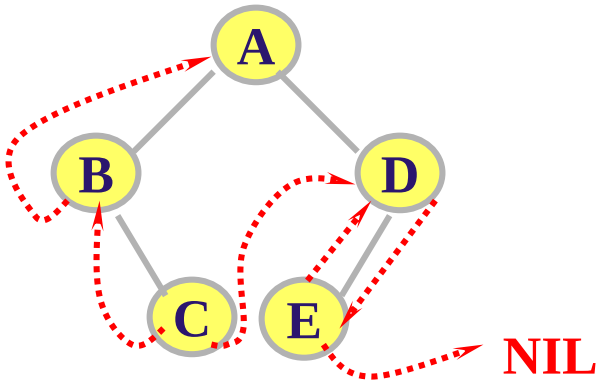
ltag= { 0 lchild域指示结点的左孩子
1 lchild域指示结点的前驱

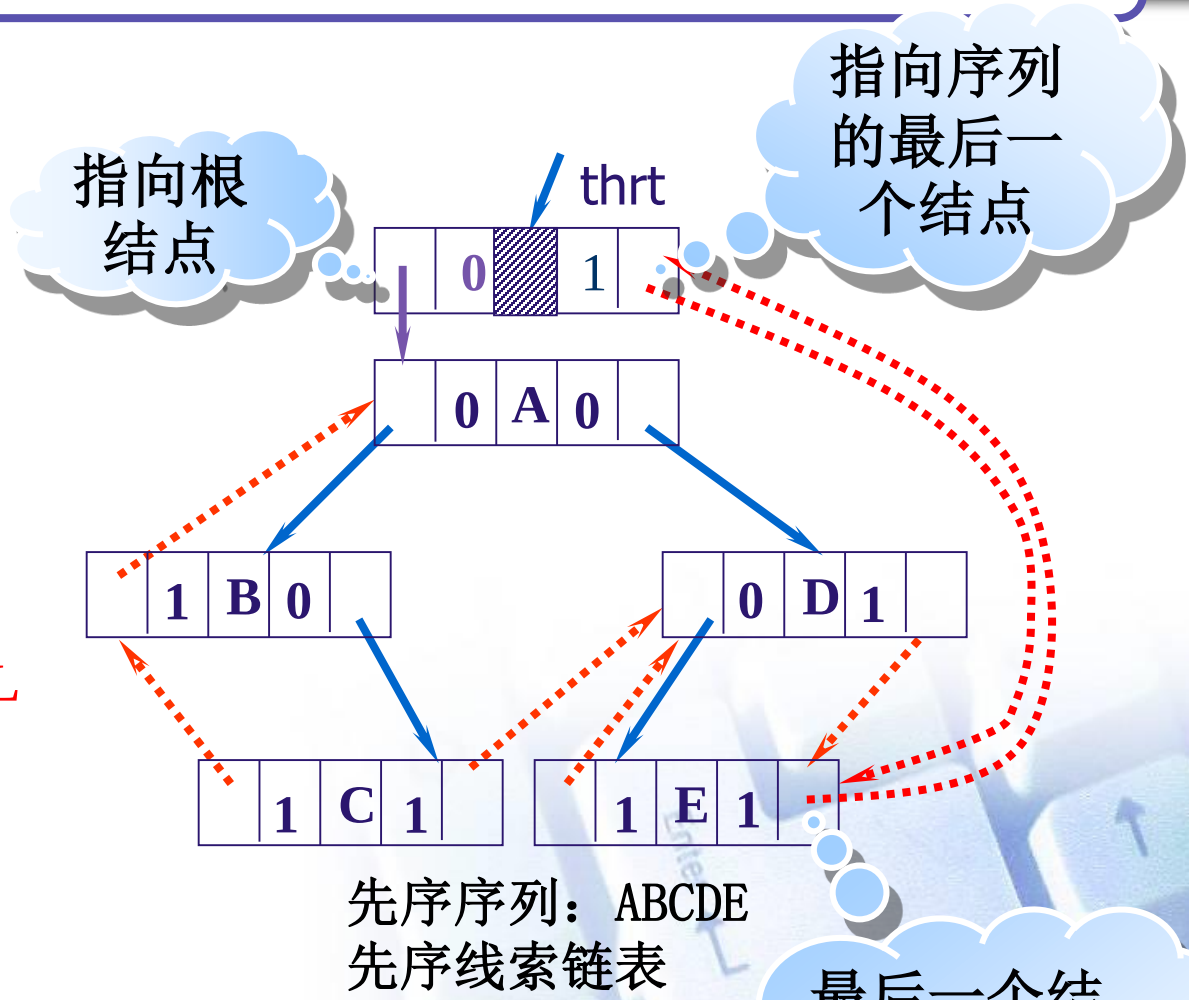
rtag= { 0 rchild域指示结点的右孩子
1 rchild域指示结点的后继

术 语

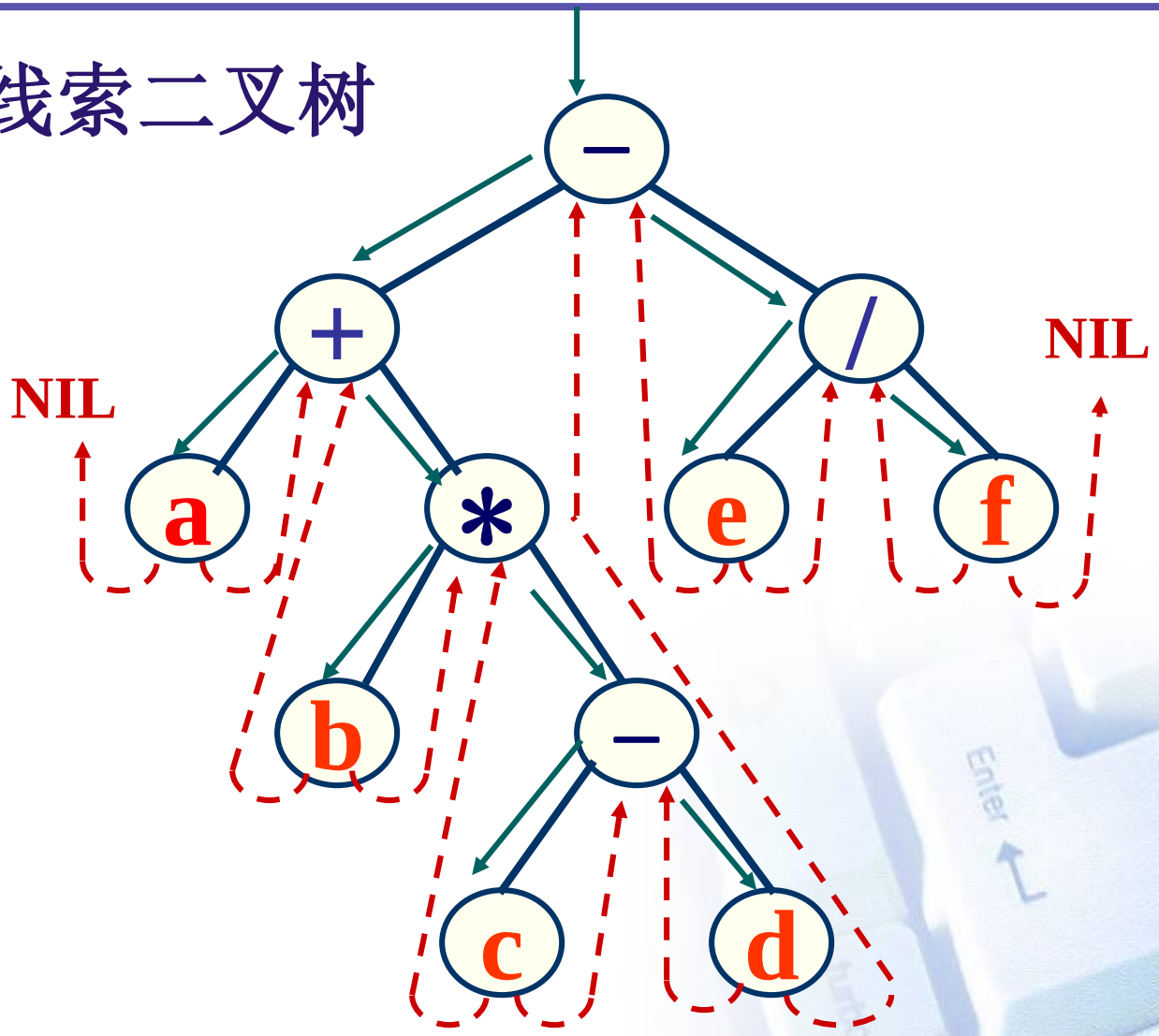
- 线 索 链 表：由上述结点结构构成的二叉链表
- 线　　索：指向结点前趋和后继的指针
- 线索二叉树：加上线索的二叉树
- 线　索　化：对二叉树以某种次序遍历使其变为线索二叉树的过程

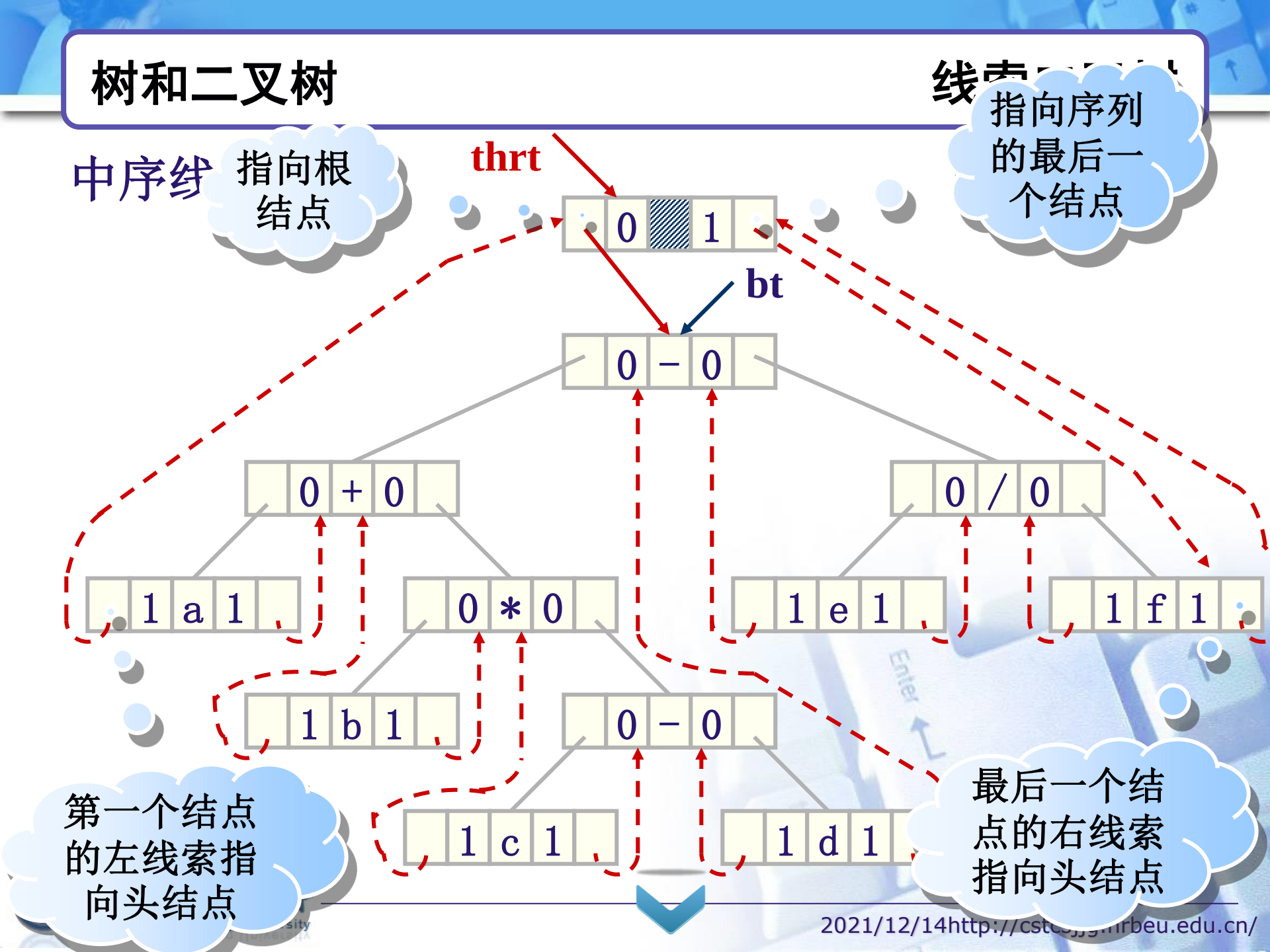
先序线索二叉树


A
B
D
C
E
NIL

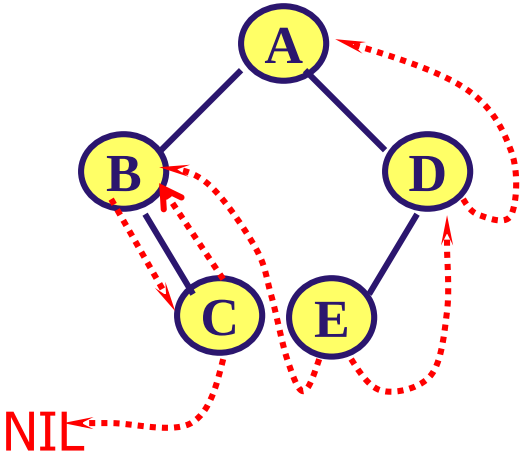

指向根结点
thrt
指向序列的最后一个结点
0 1
0 A 0
1 B 0
0 D 1
1 C 1
1 E 1
先序序列：ABCDE
先序线索链表
最后一个结点的右线索指向头结点

中序线索二叉树

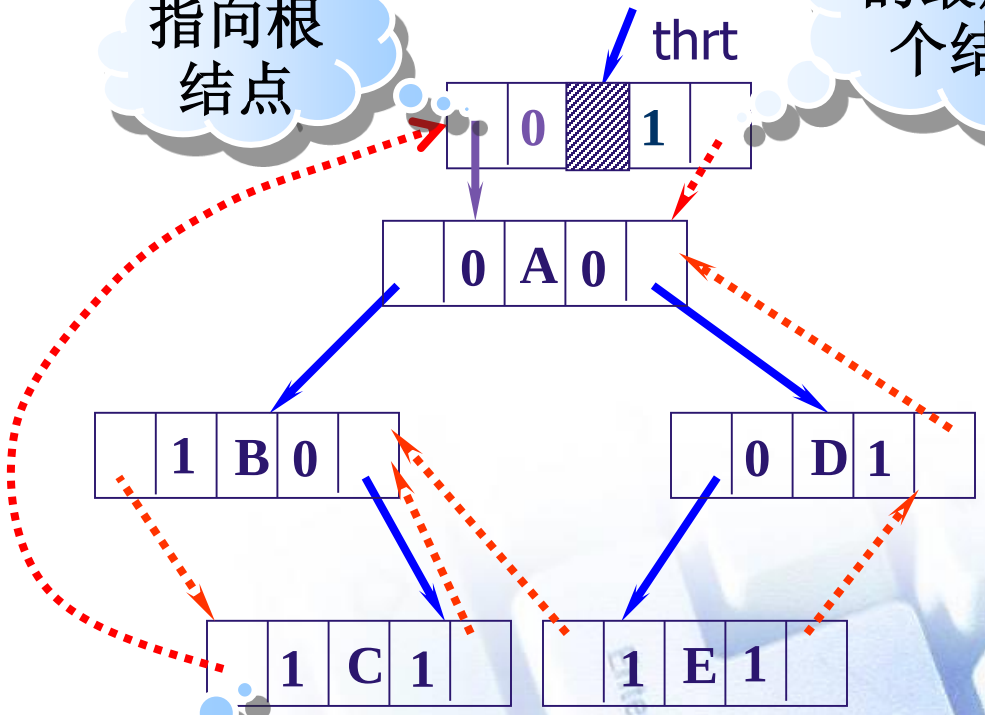



中序线
指向根结点
thrt
bt
指向序列的最后一个结点
0 1
0 - 0
0 + 0
0 / 0
1 a 1
0 * 0
1 e 1
1 f 1
1 b 1
0 - 0
1 c 1
1 d 1
第一个结点的左线索指向头结点
最后一个结点的右线索指向头结点

后序线索链表



后序线索二叉树



后序序列：CBEDA
后序线索链表

第一个结点的左线索指向头结点

- 二叉树的二叉线索存储表示

```
Typedef enum { Link,Thread } pointerTag;
                    //Link==0:指针，Thread==1：线索
Typedef struct BiThrNode {
    TElemType           data;
    struct BiThrNode *lchild, *rchild;
    PointerTag          Ltag, Rtag;
}BiThrNode, *BiThrTree;
```

- 线索二叉树的遍历

只要先找到序列中的第一个结点，依次找结点的后继直到其后继为空为止。

- 在中序线索树中找结点的后继
  - 若结点的rtag为1（叶子），则其后继为线索rchild所指结点
  - （否则）若结点的rtag为0（非叶子），则其后继为“从其右孩子沿着左链找到ltag为1的那个结点”（即右子树最左下的结点）

- 线索二叉树的遍历
  - 在中序线索树中找结点的前驱
    - 若结点的ltag为1（叶子），则其前驱为线索lchild所指结点
    - （否则）若结点的ltag为0（非叶子），则其前驱为“从其左孩子沿着右链找到rtag为1的那个结点”（即左子树最右下的结点）
  - 中序线索二叉树遍历算法思想
  - 在后序线索二叉树中查找结点的前驱和后继要知道其双亲的信息，要使用栈
  - 后序线索二叉树是不完善的

中序遍历二叉线索树T的非递归算法

关键问题

- 找到访问的第一个结点
  - 根据中序遍历的特点，第一个结点必定是“其左子树为空”的结点
  - 若根结点没有左子树，根结点即为中序遍历访问的第一个结点；若根结点有左子树，第一个结点是其左子树中“最左下的结点”。即从根结点出发，顺指针 lchild 找到其左子树直至某个结点的指针 lchild 为"线索"止，该结点为中序序列中的第一个结点

中序遍历二叉线索树T的非递归算法

关键问题

- 找到每个结点在序列中的后继
  - 若结点没有右子树，即结点的右指针类型标志 Rtag 为“Thread”，则指针 rchild 所指即为它的后继
  - 若结点有右子树，则它的后继应该是其右子树中访问的第一个结点。应该从它的右子树根出发，顺指针 lchild 直至没有左子树的结点为止，该结点即为它的后继

```
Status InOrderTraverse_Thr(BiThrTree T，Status(*Visit)(TElemType e))
{  //中序遍历二叉线索树T的非递归算法
   //T指向头结点，头结点的左链lchild指向根结点
   p=T->lchild;                                  //p指向根结点
   while (p!=T)                                  //空树或遍历结束时，p==T
      {  while (p->ltag==0)  p=p->lchild;        //沿p左链至叶子结点，中序首点
         if (!Visit(p->data))  return  ERROR;  //访问第一个结点，叶结点
         while (p->rtag==1 && p->rchild!=T) //p未指向最后一个结点
             {
                p=p->rchild;  Visit(p->data);    //访问后继结点
             }
         p=p->rchild;          //p指向当前层的右结点，右子树的根，右子树不
                               //为空，重新变为新二叉树，再去遍历其左子树
      }
   return  OK;
}//InOrderTraverse_Thr
```

- 中序遍历线索化二叉树的算法

  线索化是遍历过程中将二叉链表中空指针改为线索

  - 中序遍历线索化的算法思想
    - 在二叉树的线索链表上加一个头结点，令其lchild指向二叉树的根结点，rchild指向中序遍历时访问的最后一个结点
    - 令二叉树中序序列中的第一个结点的lchild和最后一个结点的rchild均指向头结点（类似双向链表，可双向遍历）
    - 设指针p指向当前访问结点，pre指向其前驱结点

- （以中序遍历为基础）
  - 如果p不为空
    - 左子树线索化(p->Lchild)
    - 若p所指当前结点的左子树为空，则左孩子指针指向前驱结点，建立前驱线索 （p->Ltag=1, p->lchild=pre)
    - 若前驱结点不为空，且其右孩子为空，则建立该前驱结点指向当前结点的后继线索(pre->Rtag=1, pre->Rchild=p）
    - 前驱指针后移(pre=p)
    - 右子树线索化(p->Rchild)

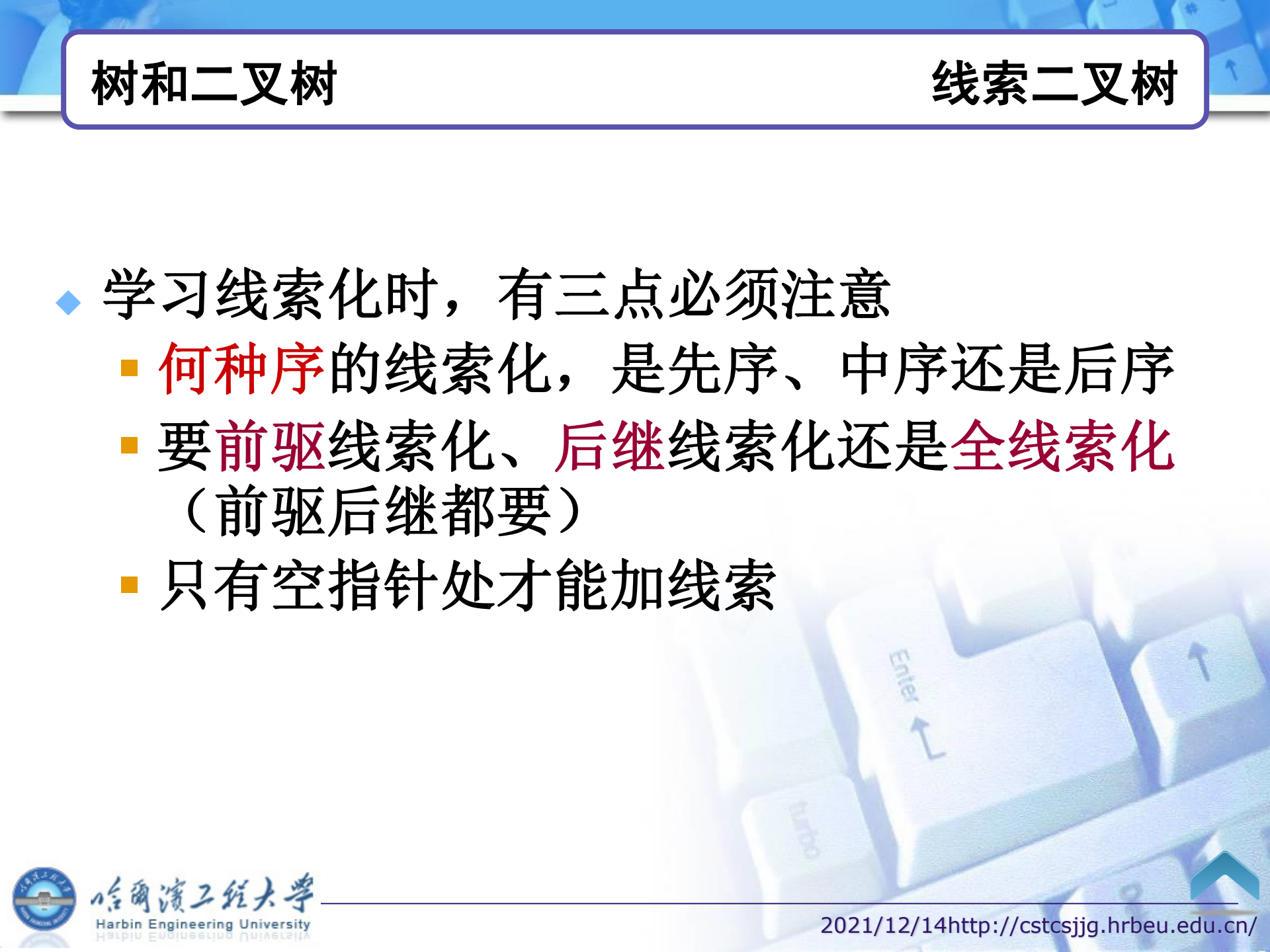
- 中序遍历进行线索化算法思想：

(1)若当前结点的左子树为空，则建立指向其前驱结点的前驱线索

(2)若前驱结点不为空，且其右孩子为空，则建立该前驱结点指向当前结点的后继线索。

```
void InThreading(BiThrtTree p)
{   //中序遍历进行线索化
    if (p) {
            InThreading(p->lchild);   //左子树线索化
            if (!p->lchild)           //左子树为空，指向前驱
                  {  p->ltag=1;
                     p->lchild=pre;
                  }
            if (!pre->rchild)         //前驱右孩子为空，前驱右指向当前结点
                  {  pre->rtag=1;
                     pre->rchild=p;
                  }
            pre=p;                    //保持pre指向p的前驱，p后移通过分支
            InThreading(p->rchild)    //右子树线索化，p后移一个位置
          }
} //InThreading
```

- 学习线索化时，有三点必须注意
  - 何种序的线索化，是先序、中序还是后序
  - 要前驱线索化、后继线索化还是全线索化（前驱后继都要）
  - 只有空指针处才能加线索



本章内容

1 树的三种存储结构

2 森林与二叉树的转换

3 树和森林的遍历

双亲表示法

孩子链表表示法

孩子-兄弟表示法

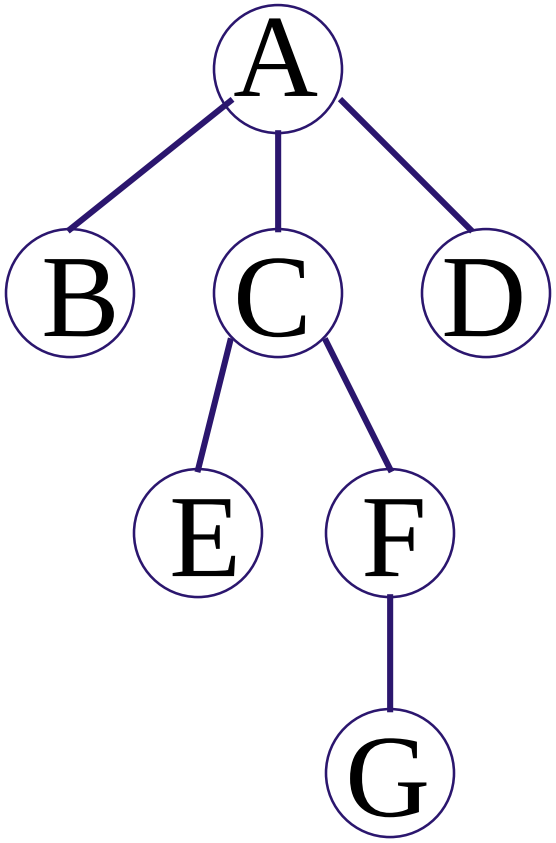
- 双亲表示法
  - 以一组连续的存储空间存放树结点，每个结点中附设一个指针指示其双亲结点在这连续存储空间中的位置
  - 结点结构

data	parent

  - 形式说明

```
#define MAX_TREE_SIZE = 100;
typedef struct PTNode{          // 结点结构
        TElemType data;
        int parent;             // 双亲位置域
} PTNode;
typedef struct {                // 树结构
        PTNode nodes[MAX_TREE_SIZE];
        int r, n;               // 根的位置和结点数
} PTree;
```

树的三种存储结构



	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	5

r=0

n=6

双亲位置

- 优点：找双亲容易
- 缺点：找孩子难，需遍历整棵树

- 孩子链表表示法
  - 把每个结点的孩子排列起来，看成一个线性表，以单链表存储；令其头指针和结点的数据元素构成一个结点，并将所有这样的结点存放在一个地址连续的存储空间里
  - 结点结构

child	nextchild

  - 形式说明

```
typedef struct CTNode {
    int         child;
    struct CTNode *next;
} *ChildPtr;
```

- 孩子链头结构

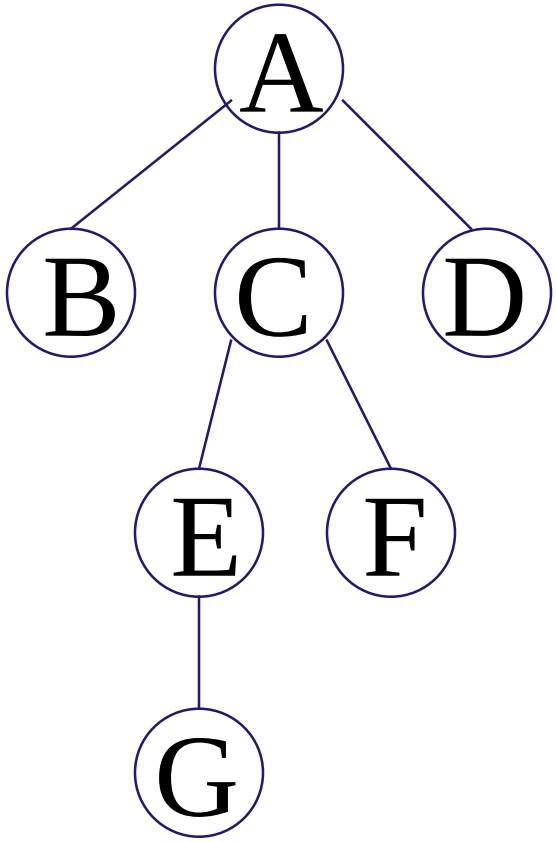
data	firstchild

- 形式说明

```
typedef struct {
    ElemType    data;
    ChildPtr  firstchild;      // 孩子链的头指针
} CTBox;
```

- 树结构
- 形式说明

```
typedef struct {
    CTBox  nodes[MAX_TREE_SIZE];
    int    n, r;          // 结点数和根结点的位置
} CTree;
```



	data	firstchild
0	A	1 → 2 → 3 ∧
1	B	∧
2	C	4 → 5 ∧
3	D	∧
4	E	6 ∧
5	F	∧

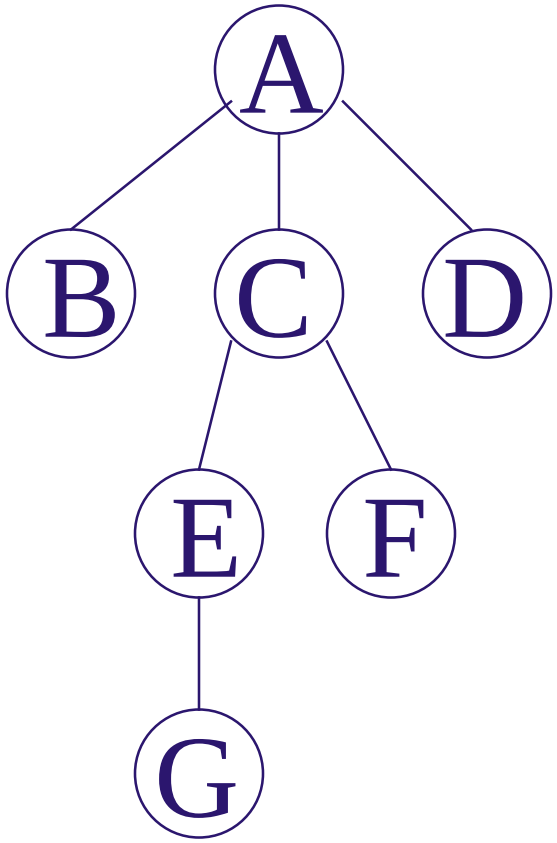
r=0

n=7

- 优点：
- 缺点：

双亲难找！！如何解决？

- 双亲、孩子表示法结合



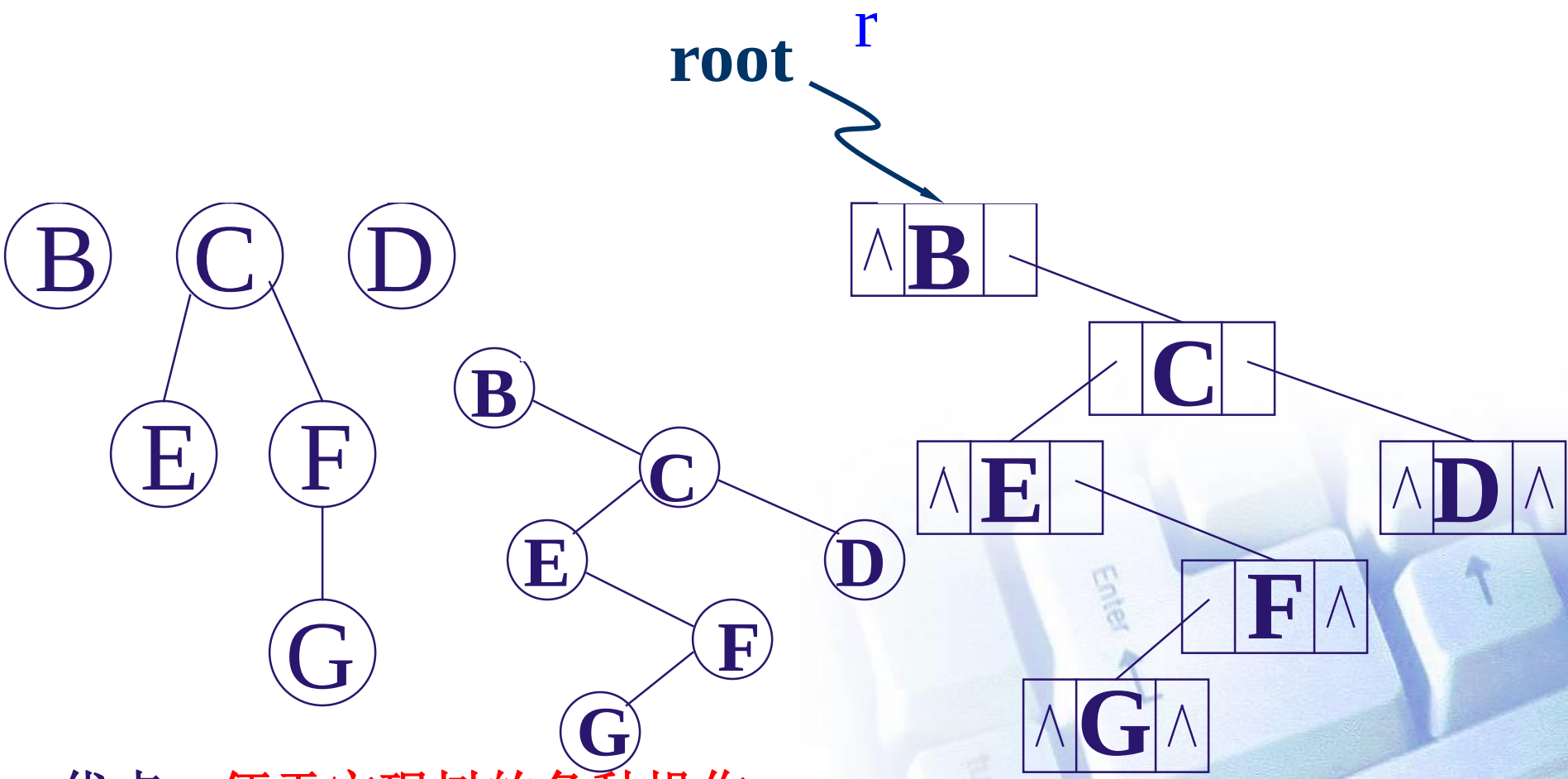
	data	parent	firstchild
0	A	-1	1 → 2 → 3 ∧
1	B	0	∧
2	C	0	4 → 5 ∧
3	D	0	∧
4	E	2	6 ∧
5	F	2	∧
6	G	4	∧

r=0

n=7

- 优点：找孩子和双亲容易
- 缺点：存储空间增加

- 孩子兄弟表示法



- 优点：便于实现树的各种操作
- 缺点：破坏了树的层次

C语言的类型描述：

结点结构：


```
typedef struct CSNode{
    ElemType      data;
    struct CSNode
        *firstchild, *nextsibling;
} CSNode, *CSTree;
```

森林和二叉树的对应关系

树和二叉树之间转换

森林和树之间转换

- 森林和二叉树的对应关系

设森林

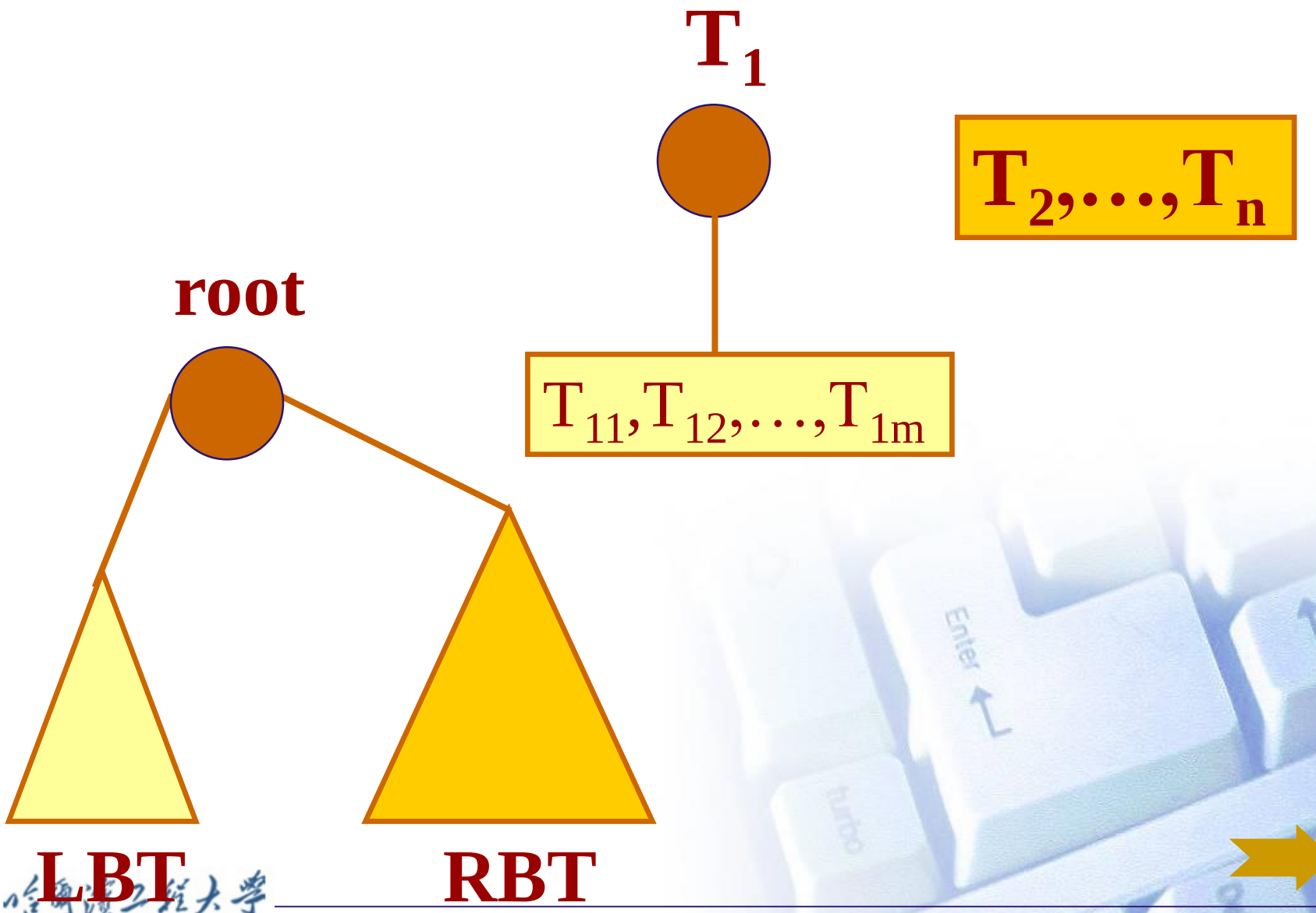
$$F = (T_1, T_2, \cdots, T_n);$$

$$T_1 = (root,\ \ t_{11}, t_{12}, \cdots, t_{1m});$$

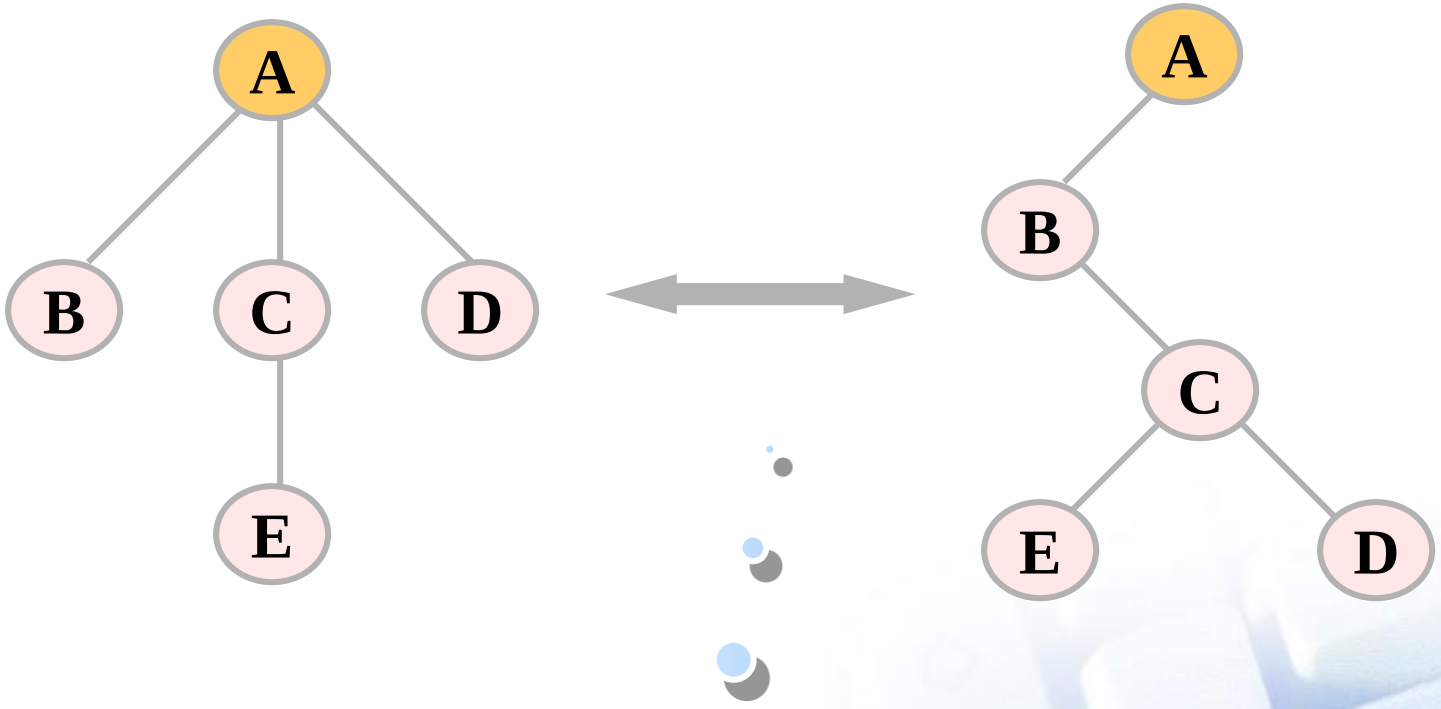
二叉树

$$B = (LBT, Node(root), RBT);$$

森林与二叉树的转换　　森林和二叉树的对应关系

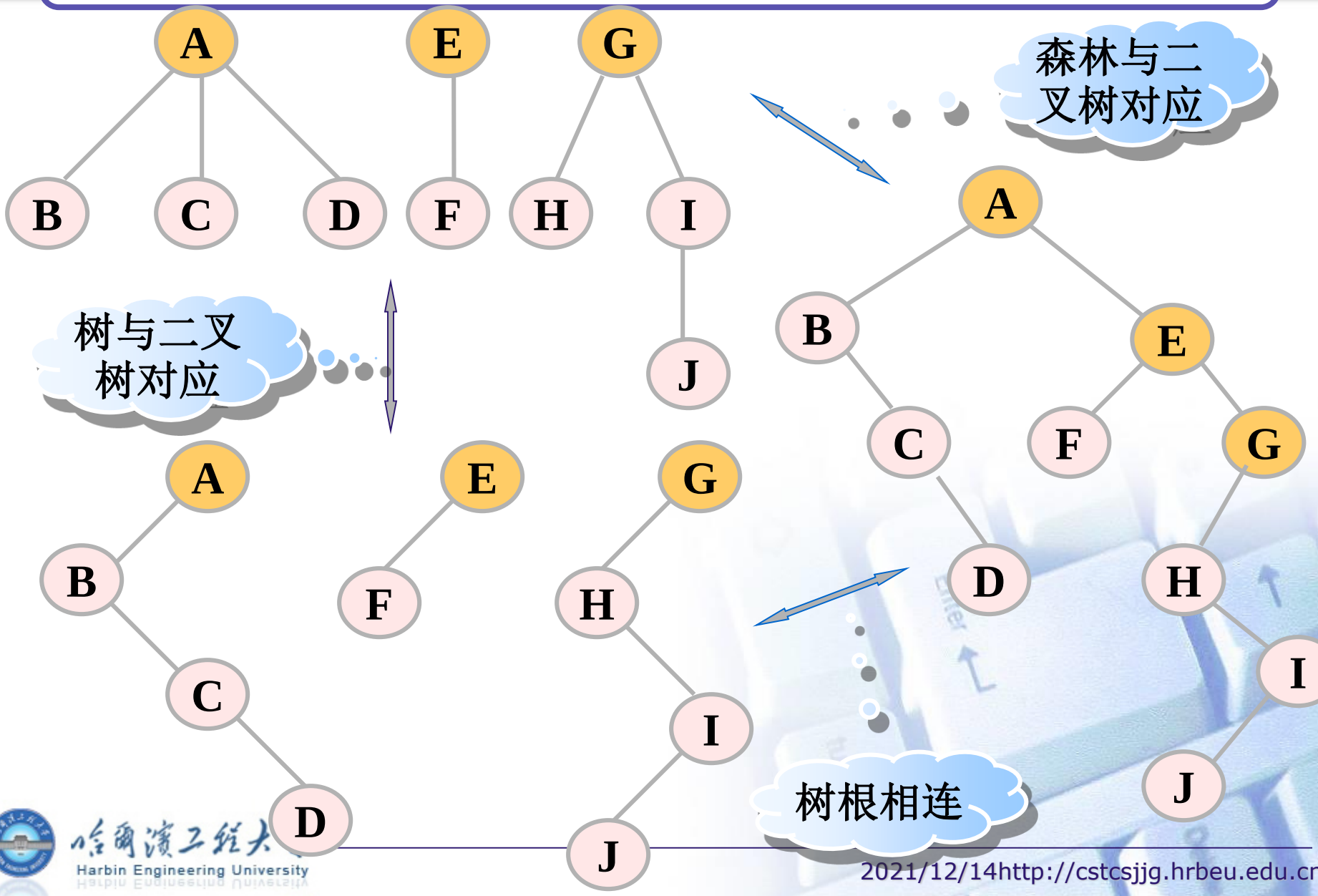


森林与二叉树的转换　　森林和二叉树的对应关系

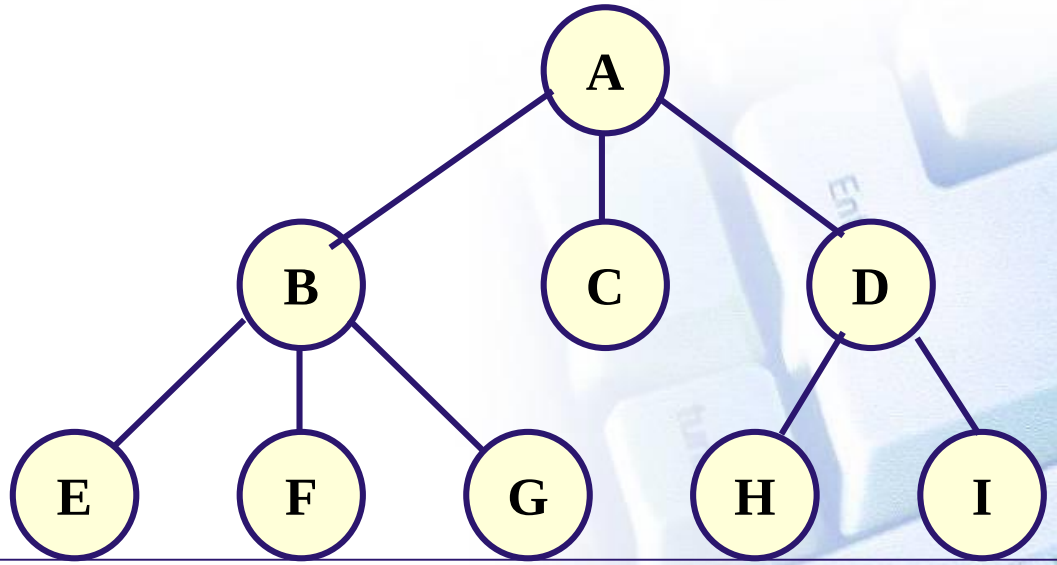


二叉链表作为
中间媒介

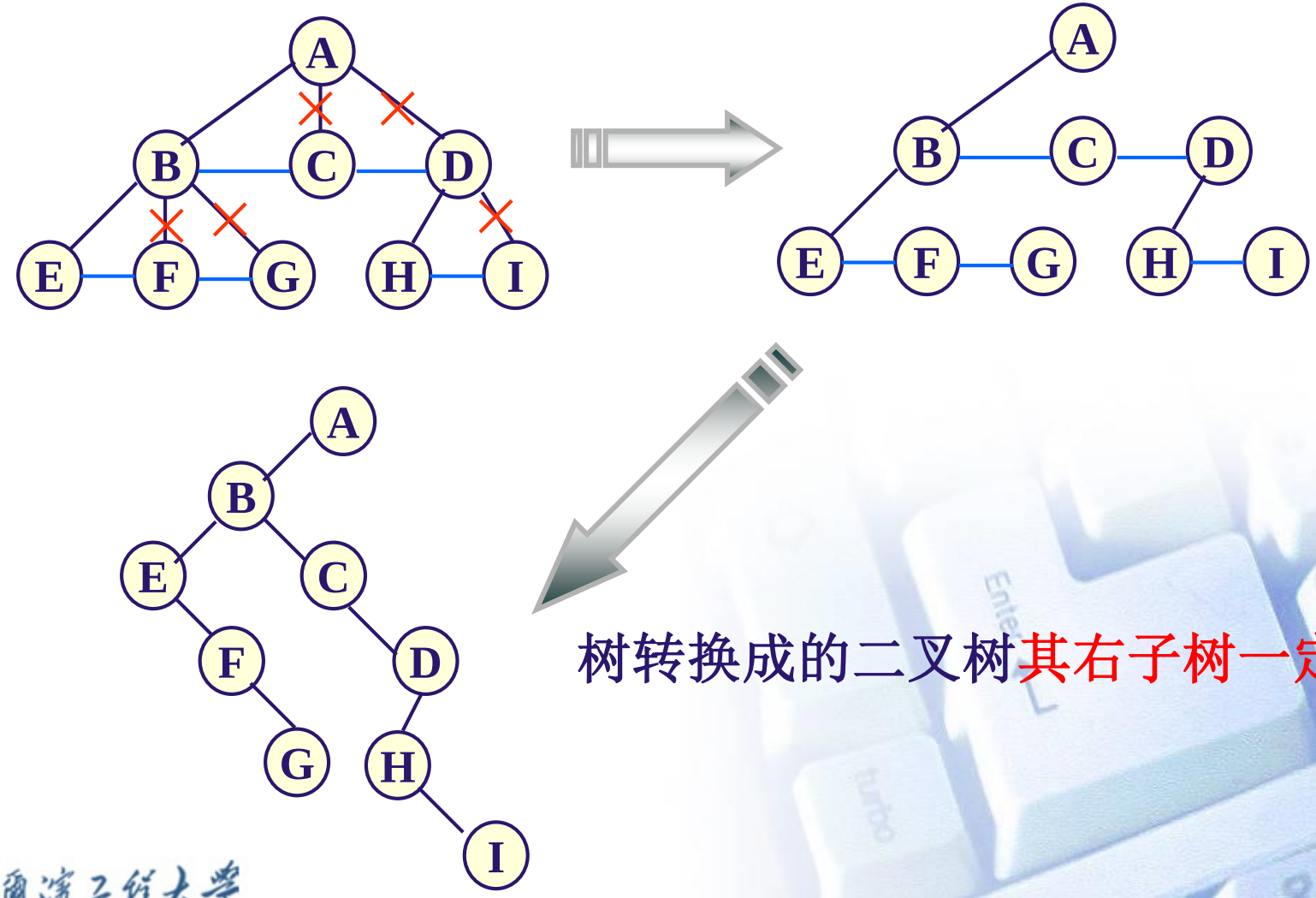
森林与二叉树的转换　　森林和二叉树的对应关系



- 树转换为二叉树的方法
  - 加线：在兄弟之间加一连线。
  - 抹线：对每个结点，除了其第一个孩子外，去除其与其余孩子之间的关系。
  - 旋转：以所有子树的根结点为轴心，将整树顺时针转 45°（加线部分旋转）

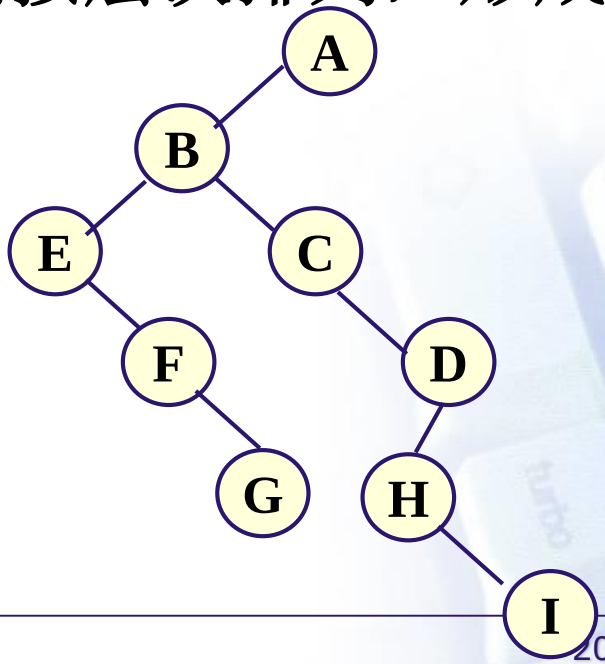


森林与二叉树的转换　　树和二叉树之间转换

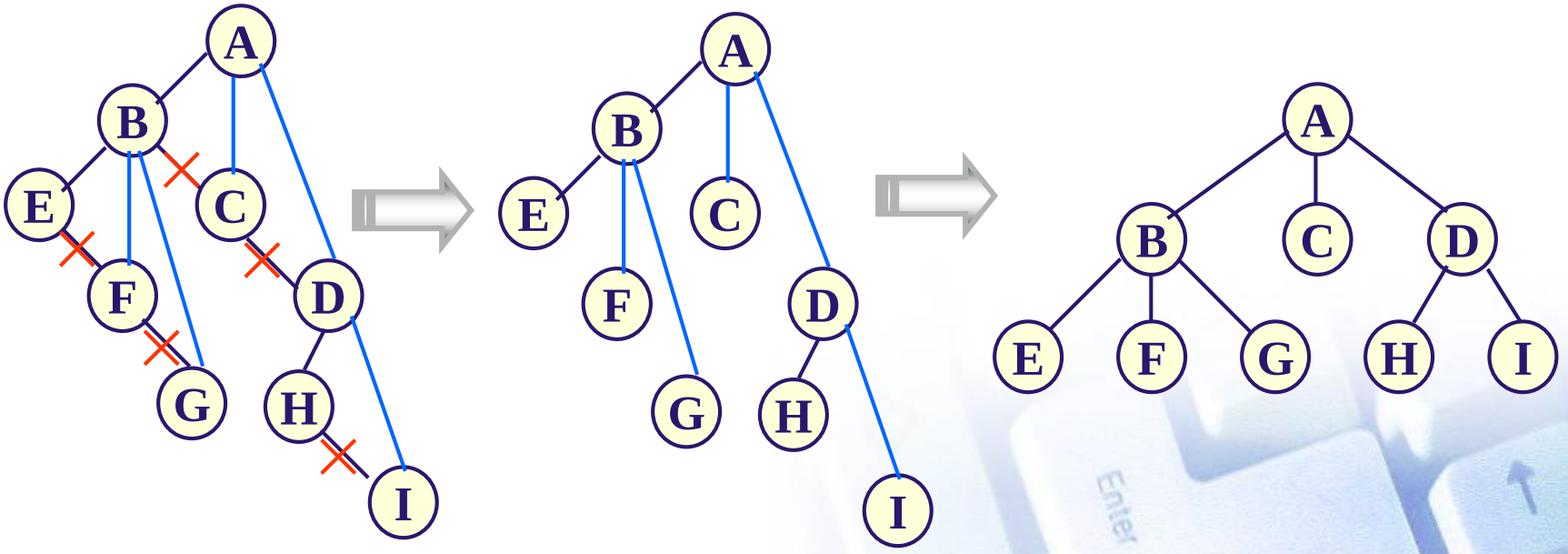


树转换成的二叉树其右子树一定为空

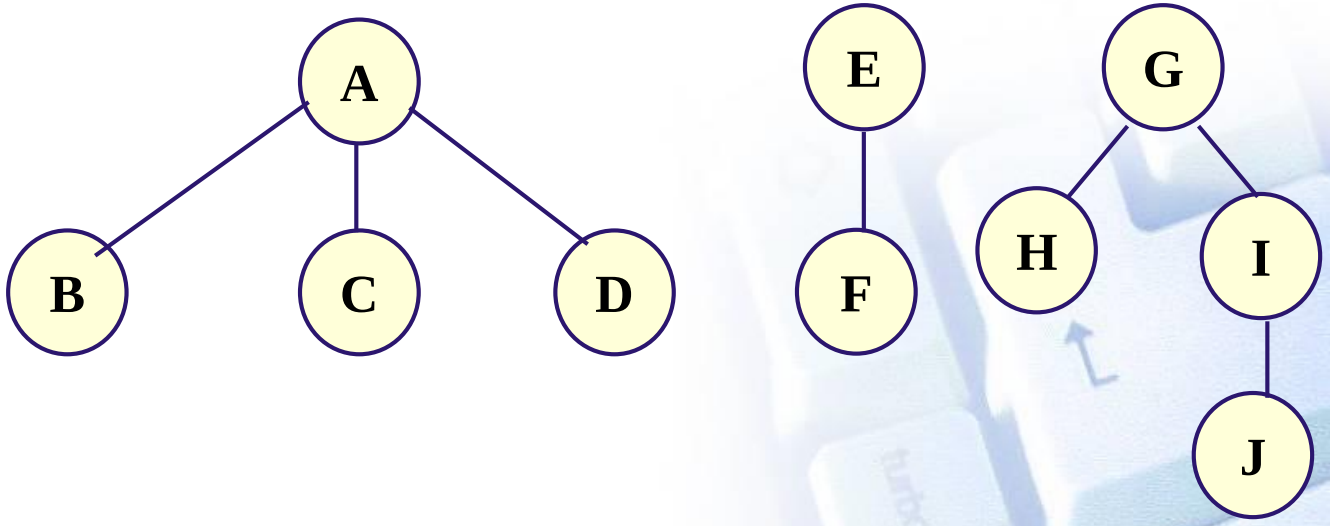
- 二叉树转换成树的方法
  - 加线：若p结点是双亲结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来
  - 抹线：抹掉原二叉树中双亲与右孩子之间连线
  - 调整：将结点按层次排列，形成树结构

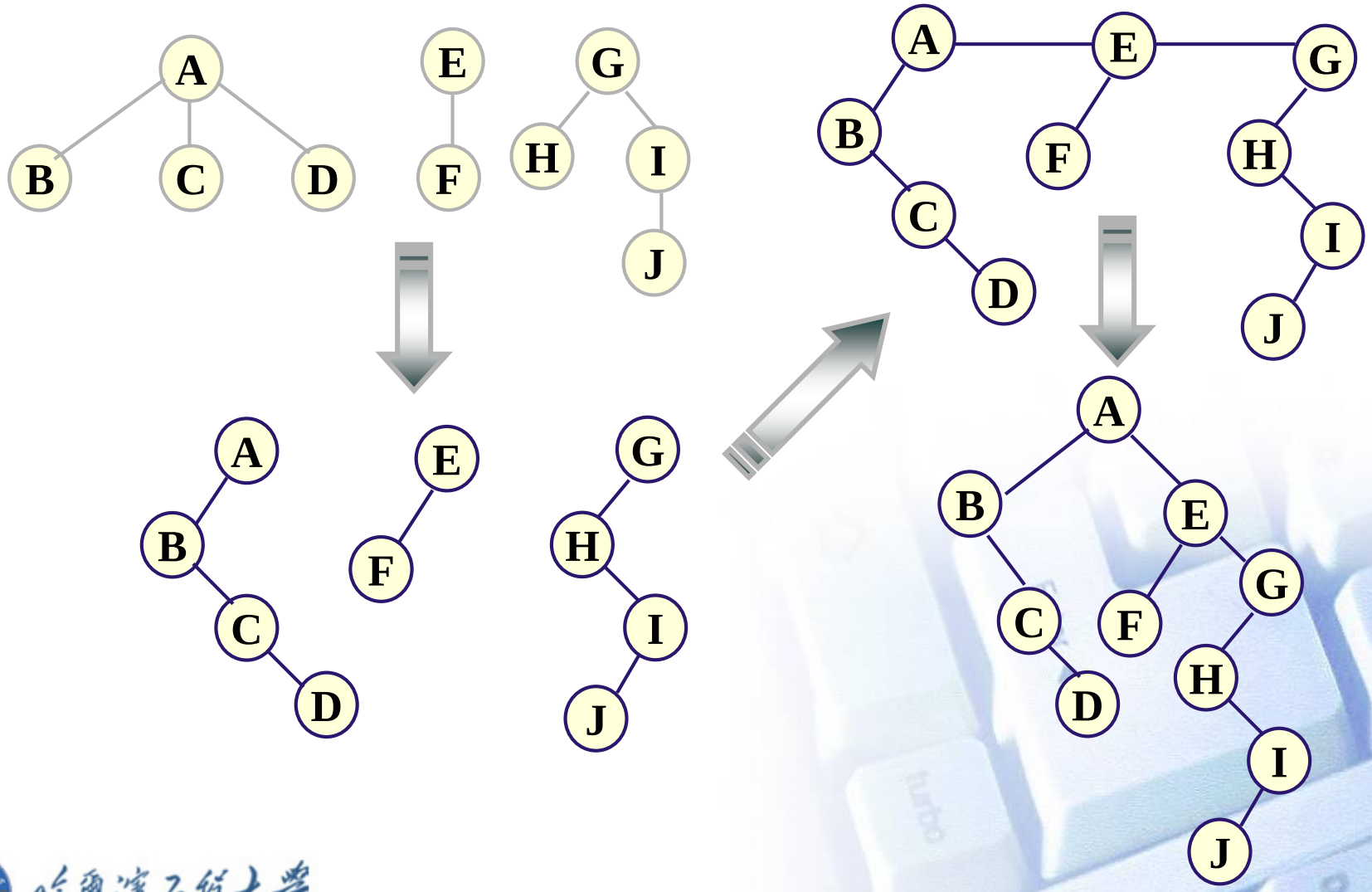


森林与二叉树的转换　　树和二叉树之间转换

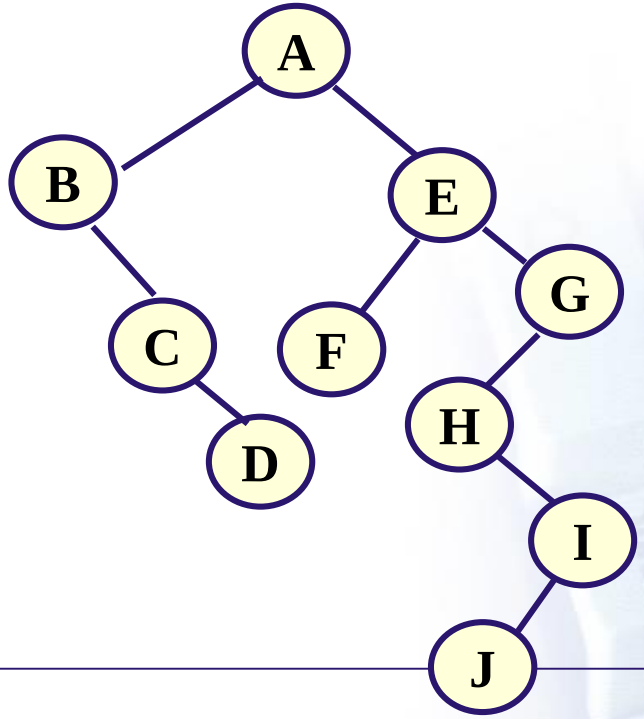


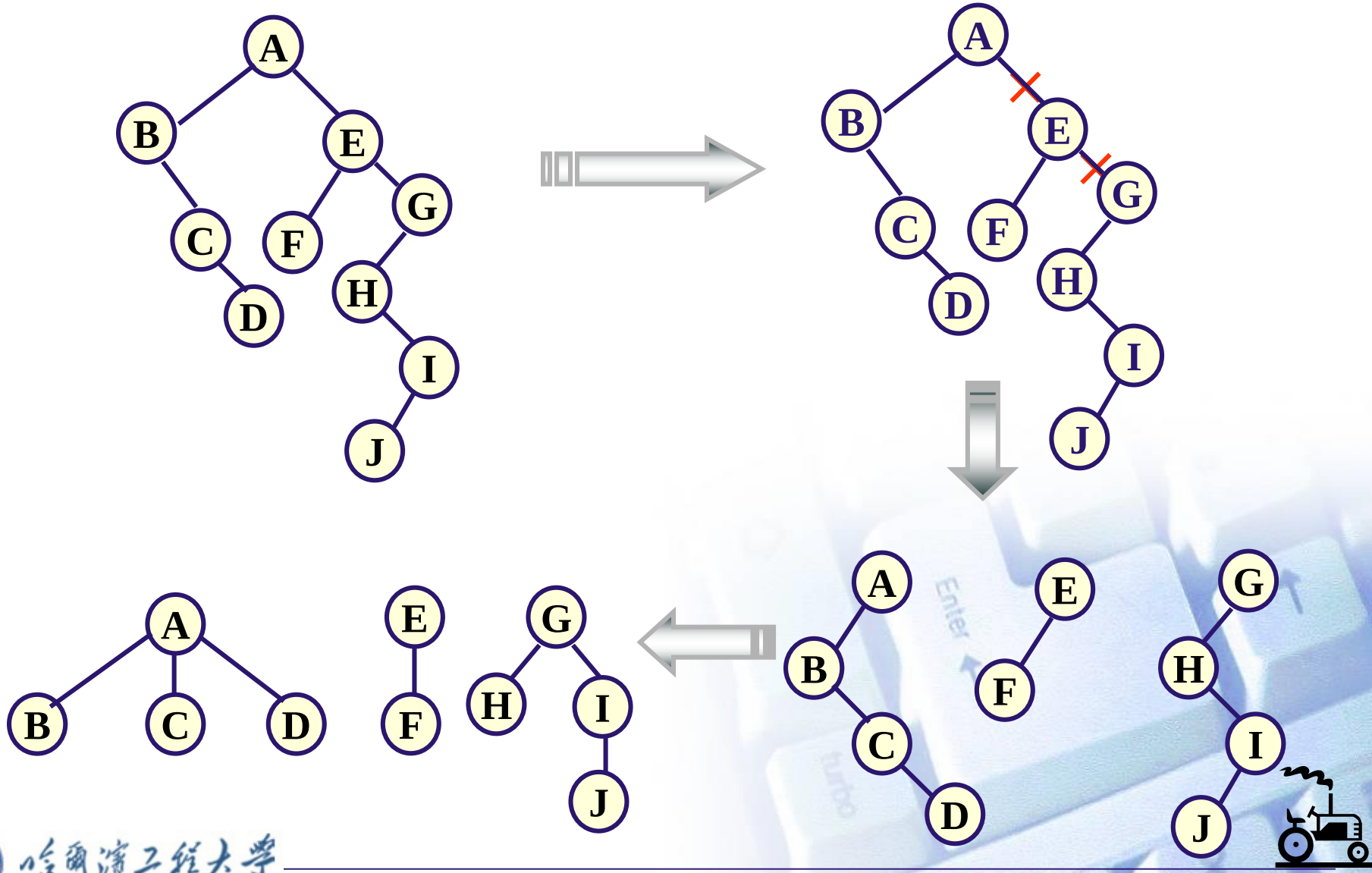
- 森林转换成二叉树的方法
  - 将各棵树分别转换成二叉树。
  - 将每棵树的根结点用线相连
  - 以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构




A
B
C
D
E
F
G
H
I
J
A
B
C
D
E
F
G
H
I
J
A
B
C
D
E
F
G
H
I
J

- 二叉树转换成森林
  - 抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树
  - 还原：将孤立的二叉树还原成树




A
B
E
C
F
G
D
H
I
J
A
B
E
C
F
G
D
H
I
J
A
B
C
D
E
F
G
H
I
J
A
B
E
C
F
D
G
H
I
J

由此，树和森林的各种操作均可与二叉树的各种操作相对应

应当注意的是，和树对应的二叉树，其左、右子树的概念已改变为：左是孩子，右是兄弟

树的遍历

森林的遍历

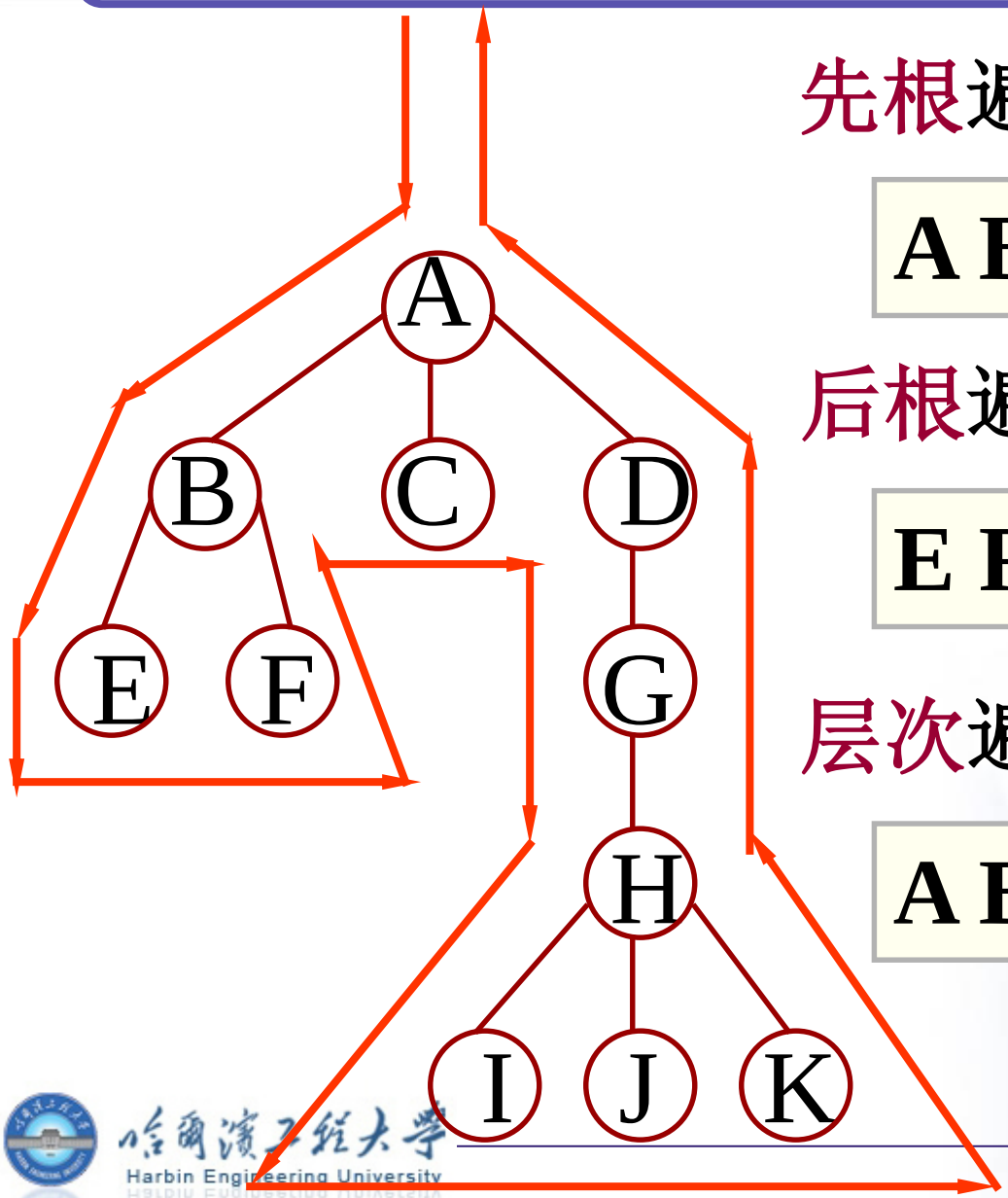
- 树的遍历
  - 先根(次序)遍历

    若树不空，则先访问根结点，然后依次先根遍历各棵子树
  - 后根(次序)遍历

    若树不空，则先依次后根遍历各棵子树，然后访问根结点
  - 按层次遍历

    若树不空，则自上而下自左至右访问树中每个结点

树的遍历



转换二叉树的
先序遍历

先根遍历时顶点的访问次序

A B E F C D G H I J K

后根遍历时顶点的访问次序

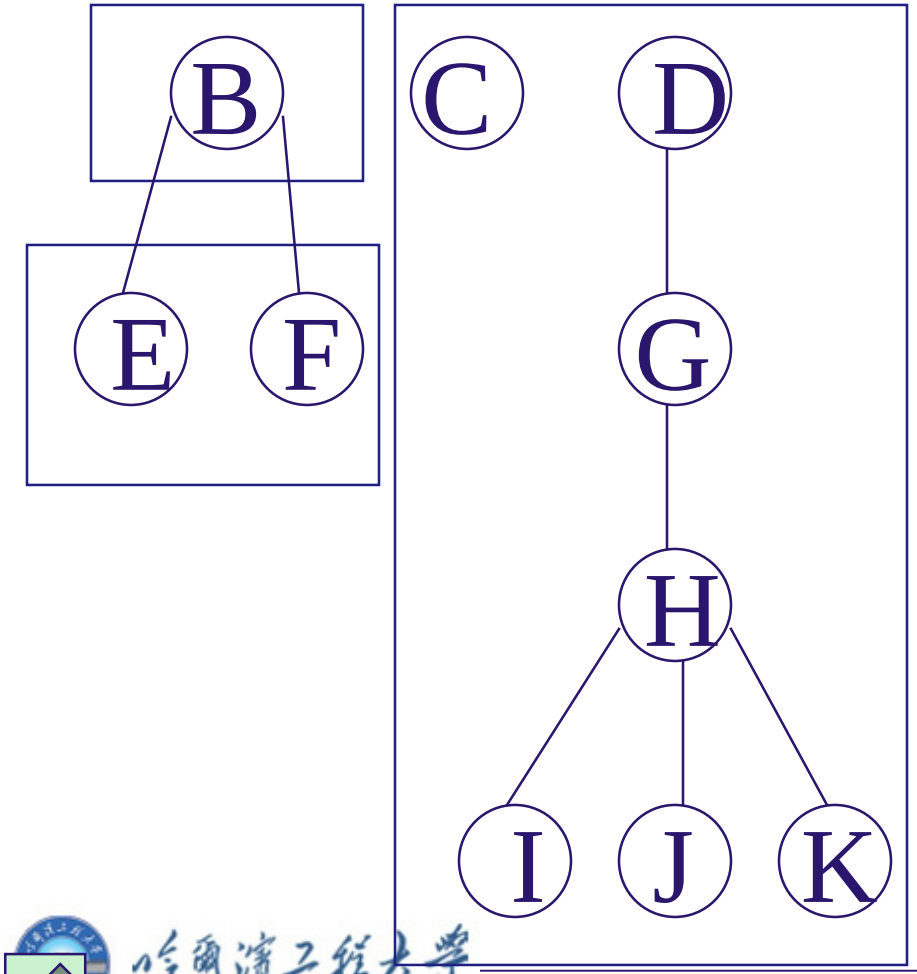
E F B C I J K H G D A

层次遍历时顶点的访问次序

A B C D E F G H I J K

转换二叉树的
中序遍历

森林 可以分解成三部分

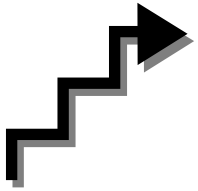


（1）森林中第一棵树的根结点；

（2）森林中第一棵树的子树森林；

（3）森林中其它树构成的森林。

- 森林的遍历
  - 先根(次序)遍历



若森林不空，则

访问森林中第一棵树的根结点

先序遍历森林中第一棵树的子树森林；

先序遍历森林中(除第一棵树之外)其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行先根遍历。

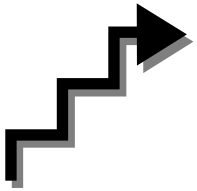
- 森林的遍历
  - 中根(次序)遍历

    若森林不空，则

    中序遍历森林中第一棵树的子树森林；

    访问森林中第一棵树的根结点；

    中序遍历森林中(除第一棵树之外)其余树构成的森林



即：依次从左至右对森林中的每一棵树进行后根遍历。

树的遍历和二叉树遍历的对应关系 ？

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

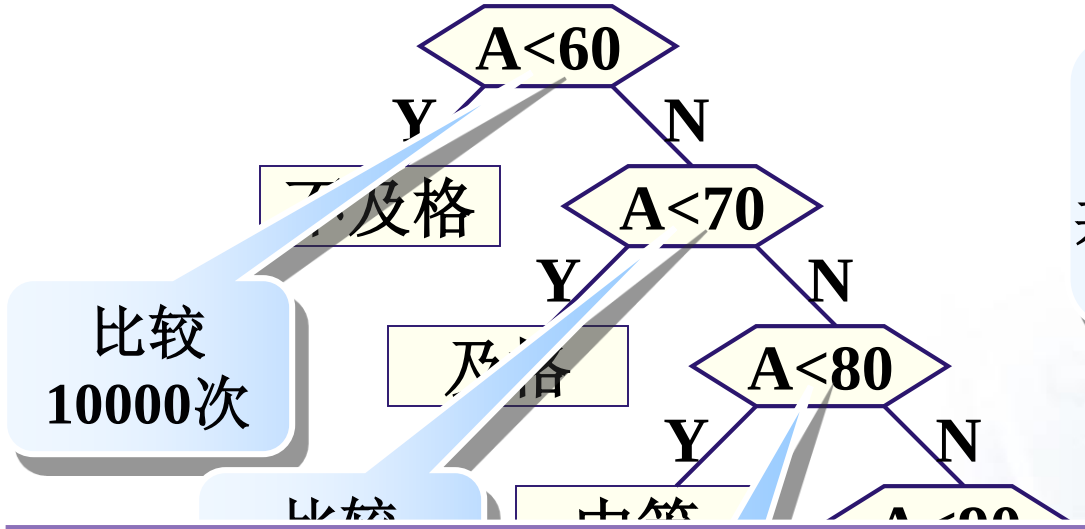
本章内容

问题提出

- **成绩统计程序问题**
  统计各分数段有多少人
- **程序的时间复杂度问题**
  程序中判断语句的比较次数
- **电报编码问题**
  编码前缀不同

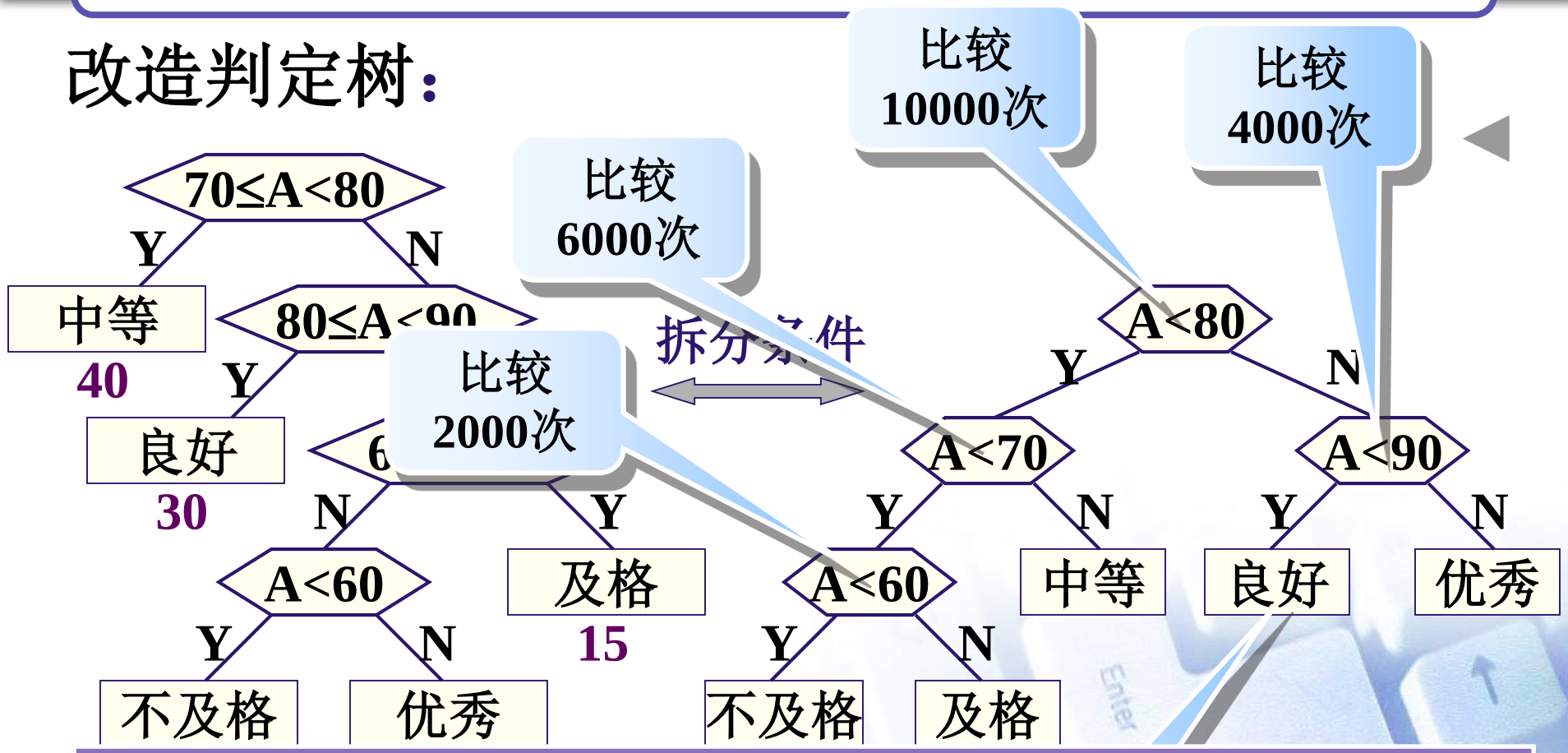
- 问题一：解决判别次数问题(输入10000个)

分数	0-59	60-69	70-79	80-89	90-100
比例数	0.05	0.15	0.40	0.30	0.10



问题：能否减少比较次数？改善算法的时间复杂度？
解决：先比较概率高的，剩下的自然就少了，比较次数则少了！！

改造判定树：



问　　题：为什么？？

特点分析：概率高的结点离根较近，概率低的离根较远！！

- 问题二：解决通信编码长短

如何对字符进行编码，使传送的电文最短??

A,B,C,D,E,F　　0,1,00,01,10,11

若 ABCDEF　0100011011　译码？字符频率？

问　　题：为什么？？
特点分析：概率高的结点离根较近，概率低的离根较远！！
解决办法：利用这个特点！！

- 最优二叉树的定义（赫夫曼树）
  - 结点的路径长度

    从根结点到该结点的路径上分支的数目
  - 树的路径长度

    树中每个结点的路径长度之和
  - 树的带权路径长度

    树中所有叶子结点的带权路径长度之和

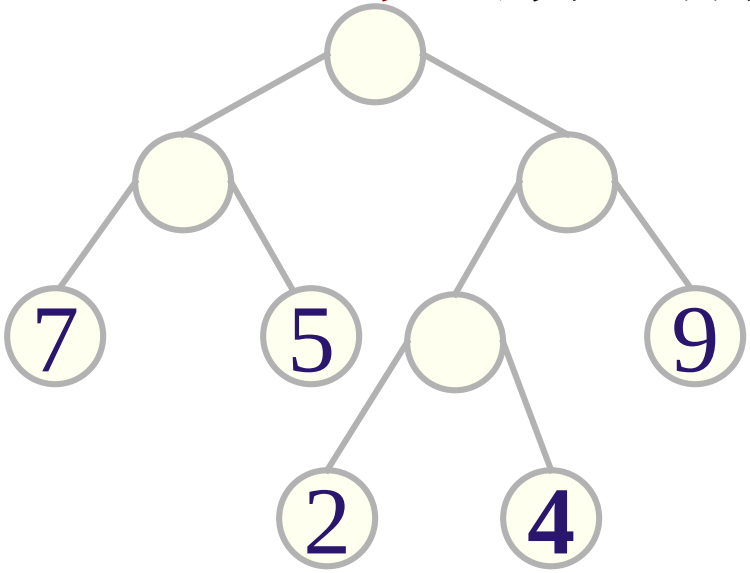
    $WPL(T) = \Sigma w_k l_k$ (对所有叶子结点)

    其中：w_k——权值

    l_k ——结点到根的路径长度

定 义

在所有含 *n* 个叶子结点、并带相同权值的 m 叉树中，必存在一棵其带权路径长度取最小值的树，称为“最优二叉树”



WPL(T)=7×2+5×2+2×3
+4×3+9×2=60

WPL(T)=7×4+9×4+5×3
+4×2+2×1=89

- 如何构造最优二叉树（赫夫曼树算法）

(1)根据给定的 n 个权值 $\{w_1, w_2, \ldots, w_n\}$，构造 n 棵二叉树的集合 $F = \{T_1, T_2, \ldots, T_n\}$

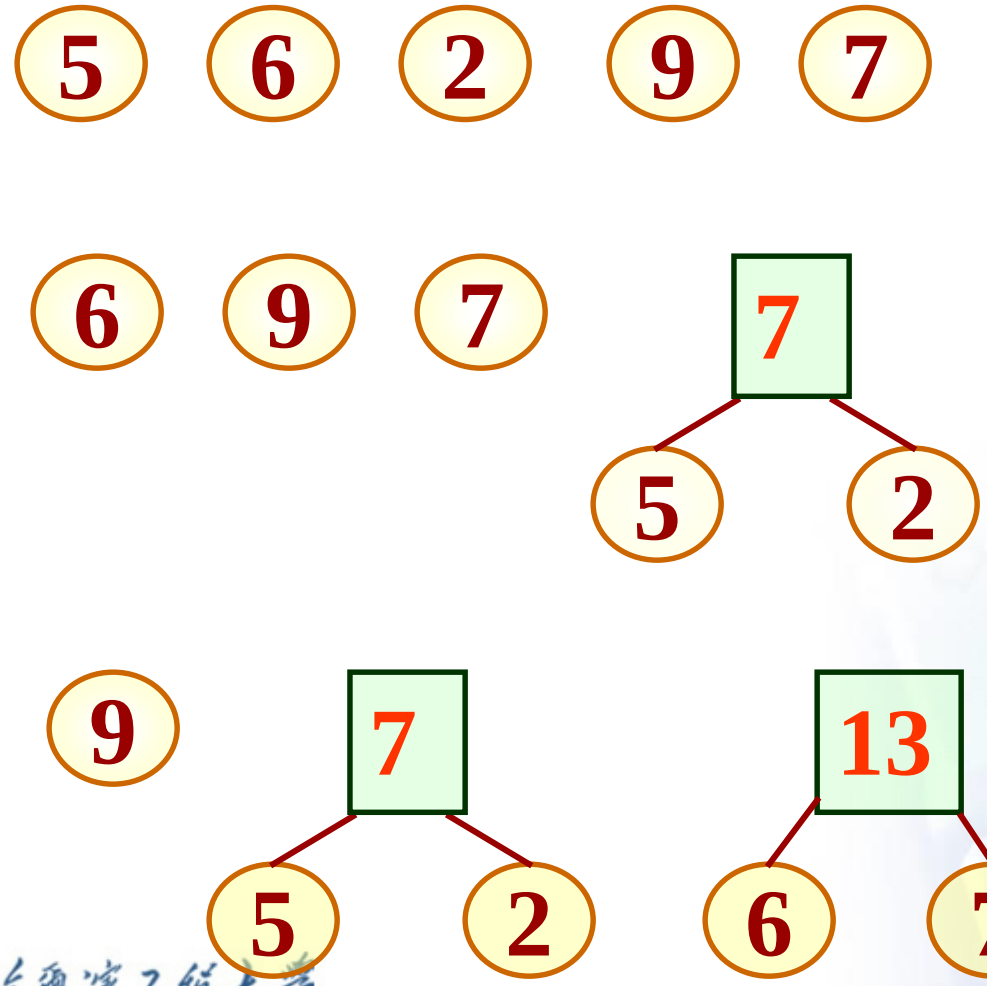
其中每棵二叉树中均只含一个带权值为 wi 的根结点，其左、右子树为空树;

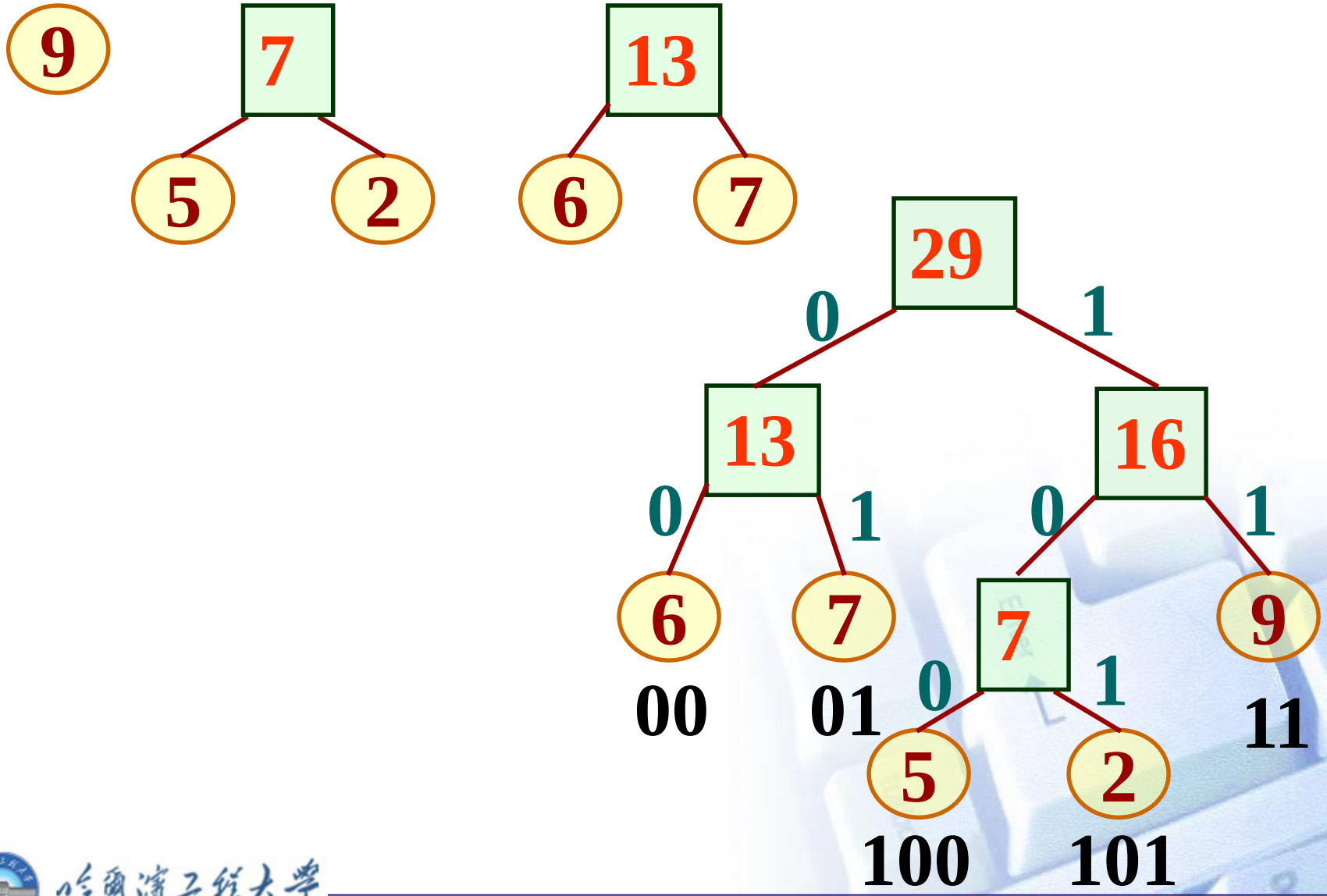
(2)在 F 中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和

(3)从F中删去这两棵树，同时加入刚生成的新树

(4)重复 (2) 和 (3) 两步，直至 F 中只含一棵树为止

例　如　已知权值　W={ 5, 6, 2, 9, 7 }




9
7
5
2
13
6
7
29
0
1
13
16
0
1
0
1
6
7
7
9
00
01
0
1
11
5
2
100
101

- **Huffman编码**
  - 数据通信用的二进制编码
    - 思想：根据字符出现频率编码，利用赫夫曼树构造一种不等长的二进制编码，使电文总长最短
    - 编码：根据字符出现频率构造Huffman树，然后将树中结点指向其左孩子的分支标“0”，指向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列

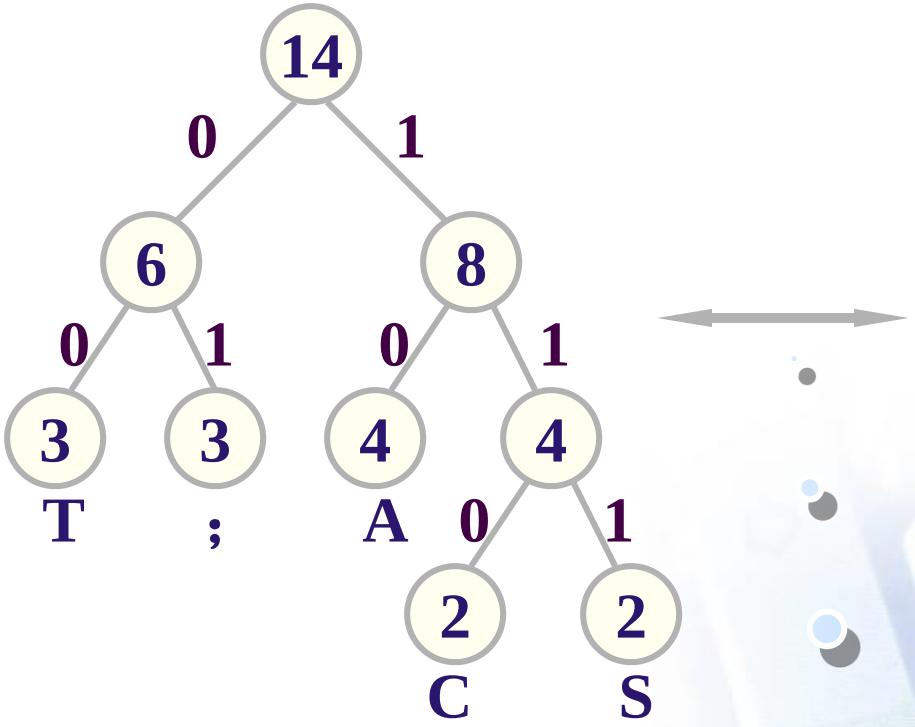
构造所得的赫夫曼编码是一种最优前缀编码，即使所传电文的总长度最短。

任意一个字符的编码都不能是另一个字符的编码的前缀，这种编码称为前缀编码。

例 如

要传输的字符集 D={C, A, S, T, ; }

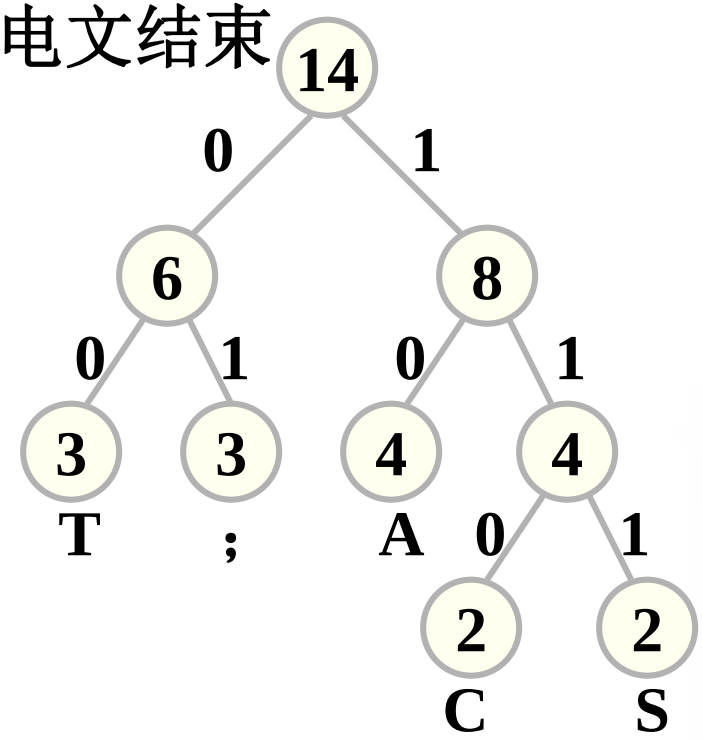
字符出现频率 w={2, 4, 2, 3, 3}



T : 00
; : 01
A : 10
C : 110
S : 111

编码从根结点到叶子

- 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



T： 00
；： 01
A： 10
C： 110
S： 111

电文为“1101000”
译文只能是“CAT”

例 电文是{ CAS; CAT; SAT; AT }
其编码“11010111011101000011111000011000”

- **Huffman编码算法**

赫夫曼树和赫夫曼编码的存储表示

```
typedef struct
    {  unsigned int weight;
       unsigned int parent,lchild,rchild;
     }HTNode,*HuffmanTree;
          //动态分配数组存储赫夫曼树
Typedef char *HuffmanCode;
          //动态分配数组存储赫夫曼编码表
```

算法中所用变量说明：

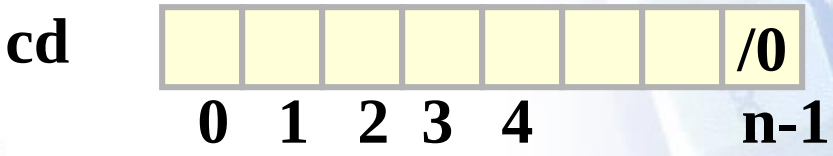
HT[]：数组，每个分量：[2n-1]（n+n-1）
- 权weight　HT表示哈夫曼树
- 父parent　存放m=2n-1个结点
- 左子lchild
- 右子rchild

cd：变量
- bits 数组[n]，每个分量为0/1。
- start 表示编码位置，是bits的下标变量。

HC：n个字符编码的头指针向量，一维数组（0下标不用）

cd：中间工作空间：n个，存放编码

例如：8个叶子（权）最多深7层，编码最多长7



不使用0下标

例如前面给定的一组权值 $w=\{7, 5, 2, 4\}$

HT

		weight	parent	lchild	rchild
p →	1	7	0	0	0
	2	5	0	0	0
	3	2	0	0	0
n	4	4	0	0	0
	5	0	0	0	0
	6	0	0	0	0
2n-1	7	0	0	0	0

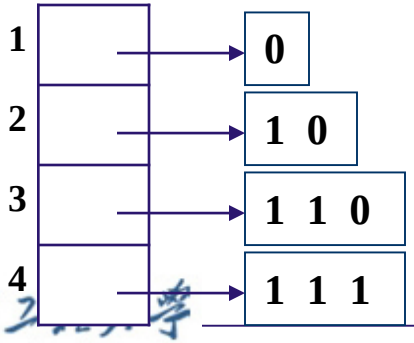
初始化赫夫曼树

HT

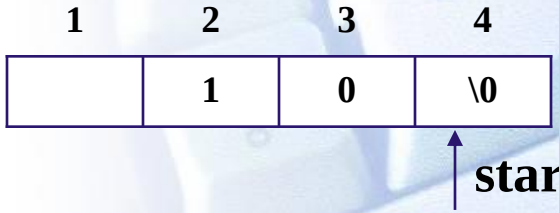
		weight	parent	lchild	rchild
	1	7	7		
	2	5	6		
	3	2	5		
	4	4	5		
i →	5	6	6	3	4
	6	11	7	2	5
	7	18	0	1	6

建赫夫曼树

HC



cd



Huffman编码算法

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{   //w存放n个字符的权值（均>0），构造赫夫曼树HT，
    //并求出n个字符的赫夫曼编码HC
   if (n<=1) return;
   m=2*n-1;                                  //n个叶子2n-1个结点
   HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode));  //0单元不用
   for (p=HT+1,i=1;i<=n;++i,++p,++w) *p={*w,0,0,0};
                                             //为HT的n个叶子赋权值
   for ( ; i<=m; ++i, ++p) *p={0,0,0,0};
                                  //HT的后n-1个结点赋值，即n个结点的未来双亲结点
   for (i=n+1;i<=m;++i)                      //建赫夫曼树
                                 //在HT[1..i-1]选择parent为0且weight最小的两个结点，
                                 //其序号（下标）分别为s1和s2
     {   Select(HT, i-1, s1, s2);             //i的值也随着赋值，逐步下移；
         HT[s1].parent=i;  HT[s2].parent=i;  //当前序号i即为s1,s2的双亲
         HT[i].lchild=s1; HT[i].rchild=s2;  //s1是i的左孩子,s2是i的右孩子
         HT[i].weight=HT[s1].weight+HT[s2].weight;//i的权是s1与s2的和
      }
```

Huffman编码算法

```
 //------从叶子到根逆向求每个字符的赫夫曼编码-------
HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
                        //分配n个字符编码的头指针向量，0下标不用
cd=(char*)malloc(n*sizeof(char));        //分配求编码的工作空间
cd=[n-1]="\0";                           //编码结束符
for (i=1;i<=n;++i)                       //逐个字符（叶子）求编码
   {  Start=n-1;                         //编码结束符位置
      for (c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)
                                         //从叶子到根逆向求编码
         if (HT[f].lchild==c) cd[--start]="0";    //倒写，正复制
         else cd[--start]="1";
      HC[i]=(char*)malloc((n-start)*sizeof(char));
                                         //为第i个字符编码分配
      strcpy(HC[i],&cd[start]);          //从cd复制编码（串）到HC
    }
  free(cd);
}//HuffmanCoding
```

本章内容

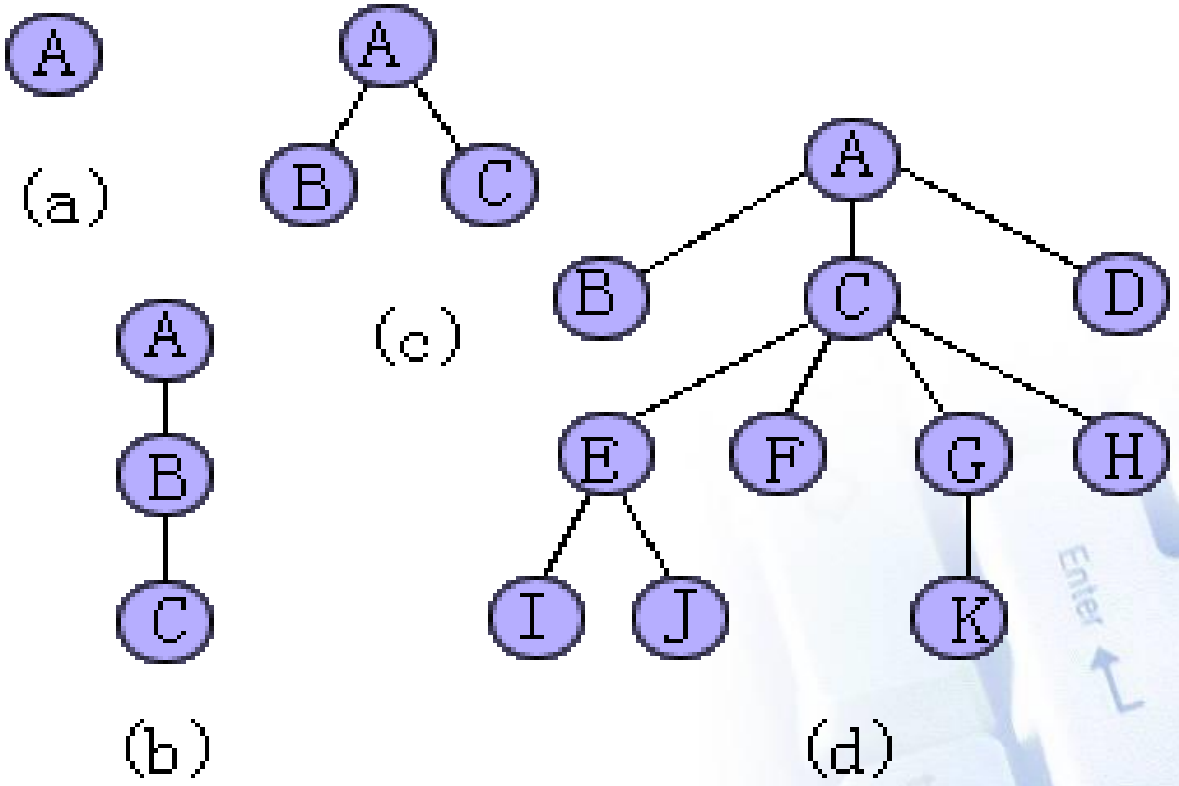
本章小结

- 树型结构中的数据元素之间存在着“一对多”的关系，它为计算机应用中出现的具有层次关系的数据，提供了自然的表示方法
- 二叉树是和树不同的另一种树型结构
- 二叉树的几个重要特性应该熟练掌握的
- 树和二叉树的遍历算法是实现各种操作的基础
- 二叉树的线索链表可看成是二叉树的一种“线性”存储结构，线索链表是通过遍历生成的
- 树和森林与二叉树的转换方法
- 哈夫曼树和哈夫曼编码的构造方法

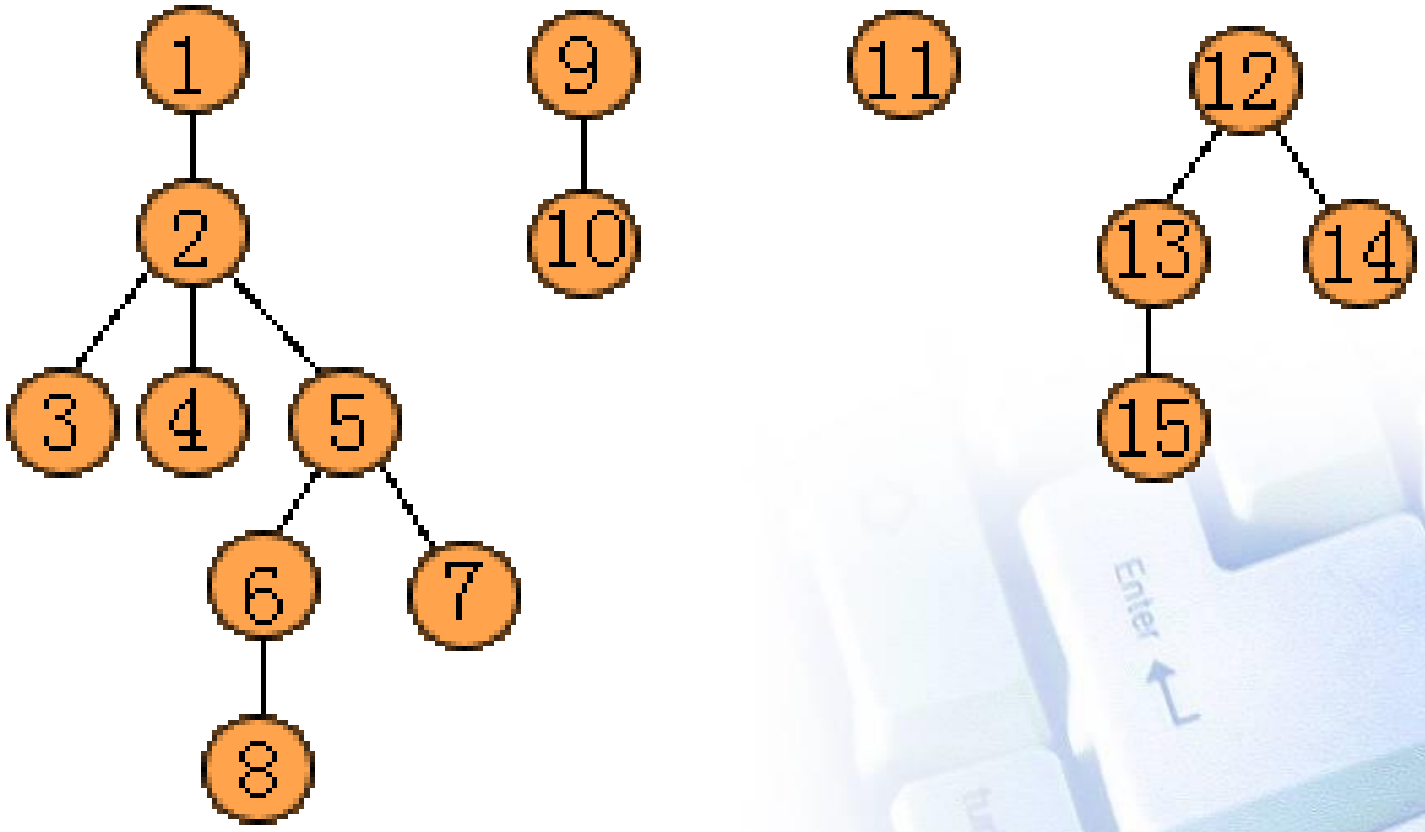
树和二叉树 基础知识题

- 试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态
- 已知在一棵含有 n 个结点的树中，只有度为 k 的分支结点和度为 0 的叶子结点。试求该树含有的叶子结点的数目。N-(n-1)/k
- 找出所有满足下列条件的二叉树:
  - 它们在先序遍历和中序遍历时，得到的结点访问序列相同;
  - 它们在后序遍历和中序遍历时，得到的结点访问序列相同;
  - 它们在先序遍历和后序遍历时，得到的结点访问序列相同;

- 分别画出和下列树对应的各个二叉树，给出各树的先根序列、 后根序列。



- 将下列森林转换为相应的二叉树



- 假设用于通讯的电文仅由 8 个字母组成，字母在电文中出现的频率分别为 0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.10。试为这 8 个字母设计哈夫曼编码。使用 0～7 的二进制表示形式是另一种编码方案。对于上述实例，比较两种方案的优缺点。
- 假设一棵二叉树的先序序列为 EBADCFHGIKJ 和中序序列为 ABCDEFGHIJK。请画出该树。
- 假设一棵二叉树的中序序列为 DCBGEAHFIJK和后序序列为 DCEGBFHKJIA。请画出该树。

- 编写递归算法，将二叉树bt中所有结点的左，右子树相互交换

```
void exchange(BiTree bt)
    // 已知 bt 为指向二叉链表(二叉树)的根指针
```

```
void exchg_tree(bitreptr BT)
{
   //采用后序遍历方法，交换每一个结点的左右子树
   if (BT){ //非空
      exchg_tree(BT->lchild); //交换左子树所有结点指针
      exchg_tree(BT->rchild); //交换右子树所有结点指针
      p=BT->lchild;          //交换根结点左右指针
      BT->lchild=BT->rchild; BT->rchild =p;
    }
}
```

下课休息一会！